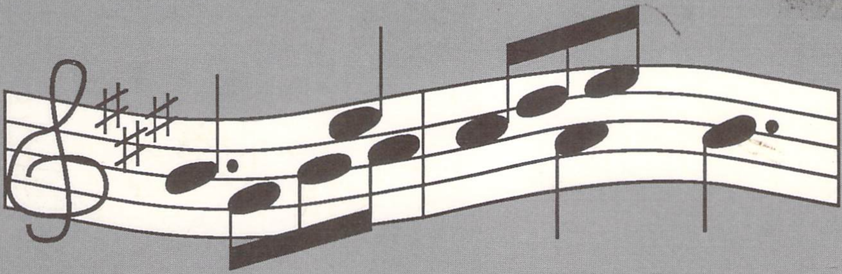
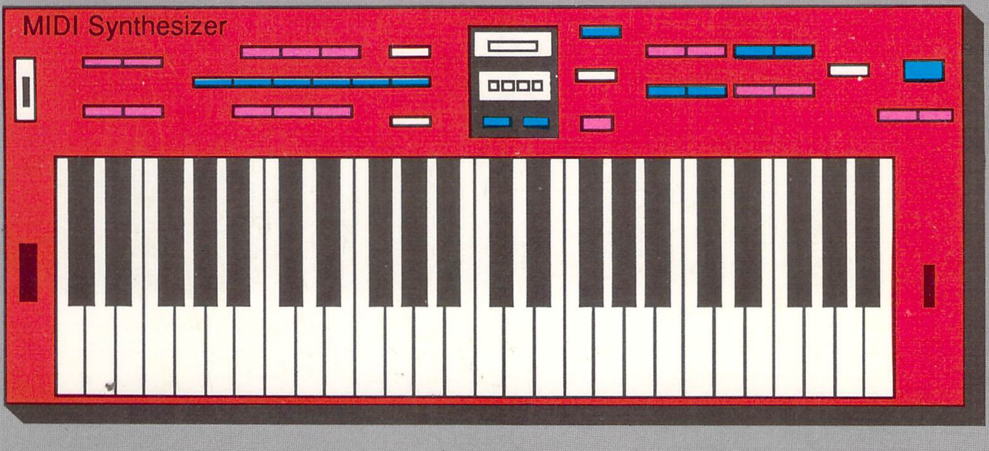
ATARI® ST™

Introduction to MIDI Programming

Explore the infinite electronic musical capabilities of the ST

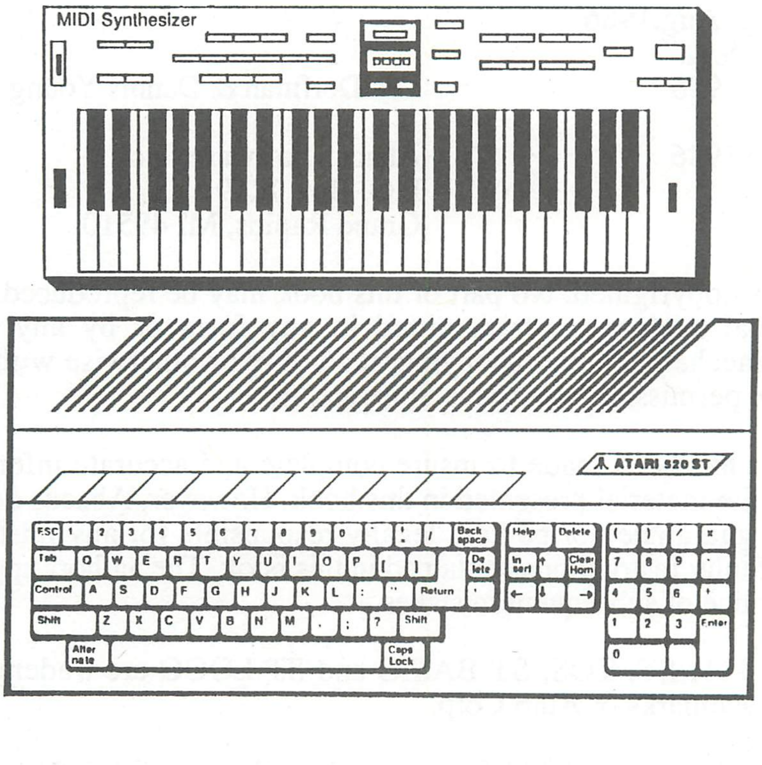




you can count on Abacus

ATARI® ST™

Introduction to MIDI Programming



By Len Dorfman and Dennis Young

Abacus Software

First Printing, Aug.1986
Printed in U.S.A.

Copyright © 1986 Len Dorfman & Dennis Young

Copyright © 1986 Abacus Software, Inc.
P.O. Box 7219
Grand Rapids, MI 49510

This book is copyrighted. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of Abacus Software.

Every effort has been made to insure complete and accurate information concerning the material presented in this book. However, Abacus Software can neither guarantee nor be held legally responsible for any mistakes in printing or faulty instructions contained in this book. The authors appreciate receiving notice of subsequent mistakes.

ATARI, 520ST, ST, TOS, ST BASIC and ST LOGO are trademarks or registered trademarks of Atari Corp.

GEM, GEM Draw and GEM Write are trademarks or registered trademarks of Digital Research Inc.

CASIO CZ-101, CZ-1000 are registered trademarks of CASIO.

ISBN 0-916439-77-1

Dedication

This book is dedicated to Barbara & Rachel Dorfman and Carol, Josh & Katy Young. This stalwart group has stuck by our sides during periods of intensely hard and chaotic work. We could not have completed this work without their patience, love and support. Thank you.

And to Michael and Linda Barnes, who believed.

Table of Contents

Preface i

Chapter I MIDI and your ST 1

1.1 Our Introduction to MIDI Programming 7
1.2 What is MIDI? 8
1.3 The Atari ST Hardware Hook-up 10
1.4 Serial Transmission 12
1.5 Qualities of MIDI SPECIFICATION 1.0 13
1.6 Programming your Synthesizer 20
1.7 A Guide For Buying your Synthesizer 22
1.8 A Guide for Buying your MIDI Software 26
1.9 Chapter 1 Summary 28

Chapter II The MIDI LANGUAGE 29

2.1 BITS & BYTES 31
2.2 Data Format 36

Chapter III Programming your Atari ST 51

3.1 Hexadecimal Notation 53
3.2 The Extended BIOS 56
3.3 `scaler.c` source code 59
3.4 `readkey.c` source code 66
3.5 `patch.c` source code 77

Chapter IV ST MUSIC BOX AUTO-PLAYER 93

4.1 In the Beginning 95
4.2 `eio.c` source code 101
4.3 `mdrive.c` source code 119
4.4 `sets.c` source code 145
4.5 `btdata.c` source code 168
4.6 `interf.c` source code 193
4.7 `move1.c` source code 233

Appendices 251
Compiling the programs 253
Optional diskette notes 254

Index 255

Preface

The Atari ST is a powerful computer line allowing both the creative programmer and the musically inclined a multitude of opportunities. One of the many fascinating features of the ST's architecture is the built-in MIDI interface and the MIDI INPUT AND MIDI OUTPUT ports on the back of the computer. The MIDI phenomenon is spreading like wildfire through the music community. This book is intended to explain what the MIDI experience is, and how the reader may use the ST to access these MIDI ports to communicate with commercially available synthesizers.

In this book we have decided to publish our source code to the AUTO-MUSIC player for XLent Software's ST MUSIC BOX. Publishing a segment of our commercial source code feels a tad like improperly exposing ourselves. One positive aspect of this act is that the ST MUSIC BOX's file format will now be open to public inspection. It is our hope that this will encourage other MIDI software developers to write file conversion code from other editors to our AUTO-PLAYER (presented in this book) and to use the file output from ST MUSIC BOX's editor for other players. Both XLent Software, and we as programmers, have always believed that programs with open architecture are good for all members of the computer community. This book represents proof of our commitment. Also, this book is our way of saying "thank you" to all the programmers who have helped us learn about the development of commercial programs. It is our hope that this book may serve as one guidepost in others' quests to expand as programming artists.

The source code is written in the 'C' programming language and we used the Alcyon version of 'C' which came with the Atari developer's kit. We believe programmers facile with any language will find use in perusing the code.

We have also made the conscious decision to let the reader into the mind and heart of our programming team. By that we mean, this book is not just a book on MIDI synthesizers and the Atari ST. It is also a book which gives a bird's eye view into the lives of independent programmers. It is a book replete with the trials and tribulations of working to build a commercial program in 'C', more than a few instructive routines for the beginning 'C' programmer, and a quick start in MIDI-land for the experienced programmer.

One other note: many operations in the 'C' programming language can be coded in very few lines. We, by choice, do not write source code so it will use the fewest lines or characters. Rather, our code is constructed so we will be able to quickly understand it months or years after not looking at it and modify it at will. So, if you are an experienced 'C' hacker, feel free to shorten or rewrite the code as you please. The major objective of our code is to get the program to do what it is supposed to do and run bug-free. Our purposes have been fulfilled by this code.

In closing, we realize that the material presented in this book might be a tad overwhelming for some of our readers, but we chose to leave none of our knowledge left unsaid. It is our sincere hope that you may have as much fun reading and learning from this book as we did in programming the MUSIC AUTO-PLAYER.

Namaste'-

Len Dorfman, Dennis Young
August, 1986

Chapter 1

MIDI and Your ST

MIDI and Your ST

The Atari ST computer line has provided an affordable range of computers with unprecedented computing power and value. We were not privy to the Atari inner decision-making process which resulted in the MIDI interface being installed as standard hardware into the ST. All we know is that it affords users of the ST uncomplicated, inexpensive access into one of the most exciting and fast-moving of fields: synthesized music.

Much MIDI software is already on the market and much more is on the way. With this software, you can enter the world of the musician. With your ST and mouse, you will be able to enter complex musical pieces and play them back precisely and crisply every time. For composers, the multi-voice capabilities of the keyboard synthesizers will allow them to experiment with their pieces with unending flexibility. Singers will be able to have their own orchestra on call for instant practice, or as a replacement at a performance. Songwriters and composers will be able to have the ST print in high-resolution graphics their scores, lyrics and all. For the musician who would like to compose directly at the keyboard, have it saved into the computer's memory, and then finely tune the notes, the capability is there.

The MIDI IN and MIDI OUT ports on the back of the Atari ST are the connecting points to the magical universe of synthesized music. Before we continue with our discussion of MIDI programming, we will digress a moment.

The task of writing music is well beyond the scope of this book, but we feel that it is important to explain some basic things that relate directly to the mechanics of using your MIDI connection/synthesizer in the music creation process. Then too, it would be good to give the uninitiated at least a brief smattering of musical notation that they have a chance of testing the pleasures that music can yield (without too much work). If you are familiar with standard music notation feel free to skip this introduction and GOTO Section 1.1.

Musical notation is the art of expressing music in written form. In modern western music this usually means putting notes on a staff. A staff in its commonly accepted form is a group of five evenly spaced horizontal lines and spaces, each having its own name. Notes are used as symbols and placed in such a fashion as to express the two elemental qualities of music, duration and pitch.

Duration

Duration is the length of time afforded to the playing of a note. The actual span of time playing the note will vary depending on the piece's overall tempo but within that framework the time spent on equal notes will always be equal. Thus, if 2 seconds is spent playing a whole note in measure one, and there is no change in tempo, a whole note in measure three will also take 2 seconds. In modern use, a whole note is the longest you can select and all other notes bear a direct time relationship to it. A half note will take half as much time to play, a quarter note will take 1/4 the time to play and so on all the way down to the shortest normally used note, a sixty-fourth note. For each note there is a corresponding rest. A rest gives us an easy means of providing a precise length of time when no sound should be made.

Dots are used in music to increase by half the length of time a note will play. This is done simply by placing the dot following (to the right of) the note. If the note is a half note, you would wind up holding it for the same amount of time you would normally expend for three quarter notes together. Occasionally, you will come across a double-dotted note. In this case, the note's value would increase first by a half and then by a quarter of its original value. A half note in this case would finish up with the value of seven eighth notes (4 + 2 + 1). Rests can be dotted or double-dotted just as notes can and yield the same result.

A tie is a curved line that connects consecutive notes of the same pitch and which has the purpose of combining the two notes into a single sound that lasts as long as the total of the two separate durations. It can be used to create duration values for a single sound that can't otherwise be had, such as seven eighth notes played as one unit: a half note tied to a dotted quarter note. The other important use for a tie is to couple two notes that are separated by a bar line.

A slur is similar to a tie except that it connects notes that are not the same. For instance, if you connected a C note and a D note you would be using a slur, not a tie.

Many MIDI instruments allow *portamento*, in which the player "carries " the sound from note to note without break - that is, sliding from one note to the next in such a fashion that the intermediate notes are passed through. This effect is most often used with the trombone or stringed instruments.

A triplet is yet another device to allow the composer to vary duration. Here three notes are performed in the length of time normally allotted for two of the same type. An example of this is that a triplet of eighth notes will last as long as a quarter note (or two regular eighth notes).

Generally, you will be moving left to right along the staff and this horizontal reading is how music time passes and the piece proceeds. The basic beat in most modern music can be counted off in very regular segments such as 1-2,1-2 // 1-2-3, 1-2-3 // 1-2-3-4, 1-2-3-4 and the like. The duration of one of these segments is described as a measure or bar. A measure is set off by a vertical line called a bar line and usually a musical piece is composed of a collection of these measures.

The number of beats in a measure is indicated by the time signature. The time signature, which is a number, is placed to the left of a double bar line that occurs at the beginning of a staff. This number is a fraction such as 3/4 that has a double meaning. The top number in the fraction yields the number of beats in each of the subsequent measures and the bottom number designates which kind of note receives the beat. This time signature is in effect unless it is superceded by another one. In 3/4 time there would be 3 beats in the measure // 1-2-3, 1-2-3, 1-2-3, etc. // and the quarter note would be the "beat note". If you add up all the separate note duration values in a completed measure it will total the value indicated in the time signature. Thus, in our example of a 3/4 time signature, you might have a combination of a half note and a quarter note, three quarter notes, one quarter note and four eighth notes or a variety of other combinations.

Tempo refers to how fast the whole piece or at least a designated section of it is played. Tempo marks like *largo, adagio, andante, moderato, allegro, presto, prestissimo* (which range from slow to fast) are all Italian words that by common practice are used to show the speed at which to play a piece. They have no absolute value and are open to considerable interpretation. You should bear in mind that tempo, no matter what it is, has no effect on how the note durations relate to each other. An eighth note will always be twice as long as a sixteenth note and a whole rest will always be four times as long as a quarter rest.

Tempo and the ability to vary it is a powerful tool in constructing a composition's "mood" or "feel". Careful use of it can greatly embellish a work and even simple pieces can be enhanced by this device, sometimes with but little effort.

Pitch

Sound is what results when vibrations in the air beat against our eardrums. When the vibrations are slow we refer to the sound as being "low" and when they are fast we call the sound "high". It is this speed of vibration that determines what we refer to as pitch. Before dealing further with this notion, a little more information about our main workspace, the staff should be helpful.

A clef is placed to the left of the double bar line and is the first symbol that appears on the staff. The clefs that are used most frequently are referred to as the G or treble clef and the F or bass clef. When two staffs are used as the work area, as they often are when writing piano music, the top staff is marked with a treble clef and the bottom staff with a bass clef. This arrangement of the two staffs is referred to as a Grand Staff and its use is very common. Ledger or leger lines are short lines that are created to accommodate notes that fall either above or below the staffs.

In musical notation, the pitch of a note is indicated by its vertical position in the workspace - the higher the note the higher the line or space that is occupied. Pitches are given the alphabet letters A through G as names and each letter name is repeated many times. The reason for this is that there is a mathematical and acoustical relationship between the notes that have the same name. For example, each C note as you go up the staff represents a sound that is vibrating at twice the frequency of the previous C note. The same is true for each family of lettered notes. The acoustical similarity is not clearly understood (at least by us) but is very real - notes lettered the same sound remarkably alike though they are, of course, higher or lower than their mates. An octave is the name given to the distance between a pitch and another having the same name.

Octaves actually have more identified pitches than the eight lettered steps. Sharps and flats are used in front of different notes to create these extra steps which brings the total number up to twelve. The distance between one of these 12 notes and its neighbor on either side is a semitone or half-step. When a sharp is used, it makes the note that follows it one half-step higher; when a flat is used, it means the next note should be played one half-step lower. Sharps and flats used in this way, along with another symbol called a natural, are collectively called accidentals. Naturals are used to cancel out the effect of a sharp of a flat which had been established in the composition's key signature.

The key signature is the group of sharps or flats to the left of the double bar line that identify the key of the composition. There are as many as 15 different major keys - each of which has a related minor key - though many of them are rarely used. Whenever a note is placed on a line or space that has an accidental on it in the key signature, the note automatically assumes the value specified. In the key of D major, for instance, all notes on F and C should be treated as sharps. In the key of B Major, all notes on B and E should be treated as flats. This is where the natural comes in. If you wish to temporarily cancel the effect of the automatic sharp or flat, you put a natural before the note.

There are a great many books on music notation available to the interested reader in almost any library. Libraries are also a fantastic resource for sheet music of all kinds if you will be transcribing pieces for MIDI implementation.

1.1 Our Introduction to MIDI Programming

One sunny afternoon in the early winter of 1986 one of the people at XLent Software called us up asking if we might be interested in writing a program for a MIDI synthesizer. We jumped at the opportunity as we had spent the better part of the two previous years writing 6502 assembly and ST 'C' desktop publishing-type projects. We both were music lovers and were eager to begin the project. What started out as a promising commercial programming venture turned into a labor of love. As we moved into the coding for ST MUSIC BOX and the AUTO-PLAYER (whose source and more is presented later in the book) our orientation changed from, "How will this product be received at computer shows or by reviewers?" to "This is our own private music creation designed by us for us. If others get pleasure from its use, wonderful; if not, no matter, we stand by the art." In a sense, our attitude might be construed as arrogant, but we share it only to let you know the depth of our feelings for the pleasure we have received from our entry into the world of MIDI through programming the ST MUSIC Box and AUTO-PLAYER, and the subsequent use and testing of our programs.

After we agreed to take the project on, we called a local music store asking if there were any ***Keyboard*** magazines containing MIDI information on the shelves. Well, to cut a long story short, all the local stores' shelves containing MIDI related magazines and books were picked clean to the bone. The more we looked the more we realized that MIDI was not a term reserved for an arcane cult of electronic music fanatics. Rather, it seemed to be a tidal wave of people from all walks of life touched by music's magic.

The next day we went out and bought two inexpensive Casio CZ-101 MIDI keyboard synthesizers. Within minutes after plugging the units in and turning them on we became mesmerized. We were done for; hopelessly smitten. We lost our sensibilities and became hooked on MIDI MADNESS!!

1.2 What is MIDI?

MIDI is an acronym which stands for:

Musical Instrument Digital Interface

Simply, MIDI is an industry standard which allows electronic instruments to interconnect with controlling devices, which in the specific case of this book, will be the ATARI ST. The Atari ST and the electronic keyboard in the simplest MIDI configuration one could have. Experienced musicians might wish to have a more sophisticated set-up. Well, fortunately for those musicians, the MIDI standard was also designed to include the connecting of multiple keyboards, auxiliary manual controllers, sequencers and drum machines.

The word Digital comes in because the electronic instruments use numbers as elements for the the MIDI alphabet. The word Interface literally means to form a common boundary between two differing entities, which in our specific case would be the ATARI ST and the CASIO CZ-101 keyboard synthesizer. In other words, if we tell the ST to send the CZ-101 synthesizer the proper numbers, the CZ-101 could Play Beethoven's Fifth Symphony. The trick, of course, is to first know the correct numbers and then how to transmit them to the keyboard synthesizer.

Before we go more deeply into explaining some of the ins and outs of the MIDI standard, it seems expeditious to discuss the importance of the MIDI standard. A few years back some of the electronic giants of the corporate music world, such as Casio, Akai, Yamaha, Roland, Moog, etc., decided that if there was an industry standard for communicating with their instruments it would have a synergistic effect on sales.

Were they ever correct. This means, excepting items which will be discussed a little later in the book, if you understand how to get your Atari ST to speak to, say the Casio CZ, you will also know how to have it speak to a Yamaha DX series of synthesizers. In fact, you will be able to hook up your ST to a Casio CZ and Yamaha DX, if you wish.

The MIDI standard might be seen in a similar fashion as the record industry standard of the long playing record. You buy your record and you can play it on a multitude of turn tables connected to dozens of high fidelity amplifiers and speakers. The market for the record producers expands because their music medium need not be machine specific. The large base of high fidelity equipment promotes a large customer base and volume sales. The volume sales means that the record producers can keep record costs low. The low cost encourages the purchase of more hardware.

In contrast, let's take a quick look at what has happened in the computer industry with the 6502 microprocessor based Atari 800, the Commodore 64 and the Apple II home computers. You can not take an Atari 800 series computer disk and load it into an Apple II and visa versa. That means that software publishers do not have the same type of hardware sales base that the record producers do. That means sales volume will be lower so product cost must be higher. I remember when I bought our Atari 800 a few years back for $1200.00. I was warned by a friend that the hardware outlay was miniscule compared to the software outlay. I almost didn't buy my Atari 800 system for that very reason.

If you tend to agree with my line of reasoning it is clear that the MIDI standard is a GIANT step in bringing the low cost creative power and joy of synthesized music into the home arena. MIDI software on the ST will work with a variety of synthesizer machines allowing users to expand their musical world beyond previous expectation.

1.3 The Atari ST Hardware Hook-up

Hardware is needed for data transmission. In computer-speak, data is another word for information, and that information is stored as numbers. Computers are VERY good at manipulating numbers.

Rather than show complex configurations for a multi-MIDI arrangement, we have decided to show the elegant simplicity of a beginner's MIDI set-up. Please don't be deceived by the words "beginner's set-up"'. In this case an inexpensive keyboard synthesizer hooked up to your ST can produce resplendently beautiful sounds. They must be heard to be believed!

If you are planning to expand your current Atari ST configuration to include a MIDI device, we strongly recommend that you start with a keyboard synthesizer. If you do not have, or wish to have, piano playing skills you can consider a miniature keyboard MIDI configuration, such as the Casio CZ-101. This powerful Phase Distortion keyboard synthesizer packs plenty of punch for the dollar. The only real drawback of this inexpensive gem is that the CZ-101's keyboard does not have full sized keys. Their reduced size might prove irksome to some. The CZ-1000 has a full sized keyboard, but then it is a bit more expensive and you might do well to check comparable synthesizers manufactured by other brands before you decide to buy.

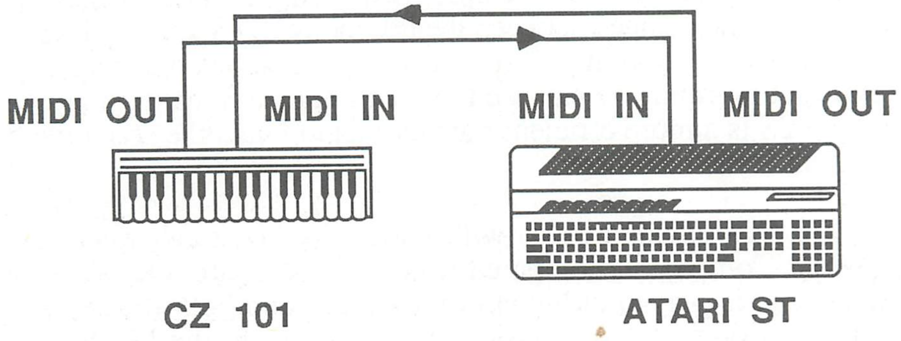
You do not have to have piano skills to use a MIDI keyboard because there are programs available right now on your Atari ST which will allow you to enter music notes into the ST's RAM memory. The programs then, using the MIDI language, can transmit the note information to the synthesizer. The synthesizer will then spring to a life of its own.

The powerful CZ-101 synthesizer has been selling of late for under $250.00 in the U.S. and in our opinion represents quite a bargain. Please understand that when it comes to computer expenditures, operate by two laws: 1) Don't spend more than what you initially need to get started! 2) Don't fix something if it's not broken. Prices in MIDI-land can range from the $250.00 U.S. dollars to figures deep into the thousands. What we are saying is that you don't have to bankrupt your savings to get high-powered performance. One other note: you will have to spend a bit more if you are interested in a more representative piano keyboard synthesizer.

Many keyboard synthesizers on the market don't have built-in amplifiers and speakers. You can hook up your synthesizer to a receiver or amplifier.

If you do not own an amplification system you will need to build or purchase one. This expense may be kept well under $120 at U.S. prices.

Now we will get to the actual hook-up. If you look on the back of the Atari ST case you will see two ports labeled MIDI IN and MIDI OUT. The Atari ST should be configured with the MIDI synthesizer in the following way:



There, that's not too complicated! The MIDI connecting cables are composed of standard 5 pin DIN connectors. As you can see from the diagram, the hook-up is very simple. You plug one end of a 5 pin DIN connector and cable into the MIDI IN port of the Atari ST and plug the other end of that cable into the MIDI OUT port of the synthesizer. Conversely, you then take the other cable and plug one DIN connector into the Atari ST MIDI OUT port and plug the other end of the cable into the synthesizer MIDI IN.

We have one other short tale about our purchasing the 5 pin DIN cables. We purchased our Casio CZ-101 synthesizers at a specialized music store in the U.S. When the salesperson asked if we were interested in special MIDI cables, we said, "Of course!" To make a long story short, they wanted over $25.00 U.S. dollars for one set of two connecting, supposedly specialized, cables. Instead, we decided to purchase the cables at a local electronics store and payed about $5.00 for one set of very nice connecting cables (saving double the price of this book for two pair, we might add). The 5 pin DIN MIDI cables are standard electronic ware and best bought in a store that caters to the electronic trade.

1.4 Serial Data Transmission

The purpose of the hardware hook-up is to allow the Atari ST and the electronic synthesizer a pathway to transfer data between each other. The language used is composed of electrical impulses. In today's data transfer technology there are two widely used methods: Parallel Data Transmission and Serial Data Transmission. In Chapter II we will discuss, for those new to computers, the differences between the infamous BITS and BYTES. For the purposes of this section, it will suffice to say that 8 bits make up 1 byte. Having that fact in mind, we can see that parallel data transmission which transfers BYTES is a more efficient transfer method than the serial method which transfers 1 bit at a time.

The MIDI creators decided to go with the bit by bit serial transmission method. Why? Computers, compared to human time, are VERY, VERY fast. Since music is played at human time speeds and both the serial and parallel methods of data transmission are fast enough, the MIDI people decided to go with the cheaper method. Serial is cheaper than parallel data transmission because there are fewer connecting wires needed for serial data transmission and serial connectors are cheaper than parallel connectors. Also, it was theorized that some nasty grounding problems would be avoided by avoiding the parallel method of data transmission and going with the serial method. The cheapest method of data transmission means lower product costs and the consumers and electronic manufacturers benefit alike. A wise decision.

For those familiar with computer communications we offer the following comparison. One industry standard data transmission rate is the well known 300 baud. The rate descriptor, 300 baud means this: 300 bits of information are being transferred every second. The MIDI standard calls for 30,000 bits to be transferred every second. In other words, the MIDI transmission rate is 100 times faster that one industry standard. Needless to say, the MIDI serial transmission rate is plenty fast for its purpose!

1.5 Qualities of MIDI SPECIFICATION 1.0

Although the MIDI SPECIFICATION 1.0 document is not massive in length, it is quite deep in nature. In an effort to present the complexity of the MIDI SPECIFICATION 1.0 in as clear a fashion as possible, we have decided to break the specification into two sub-categories. They are:

MIDI STANDARD and MIDI LANGUAGE.

The topic MIDI IMPLEMENTATION is a response to the MIDI STANDARD and MIDI LANGUAGE. In the rest of this section we will continue to refine these categories. We will do this at length because they will appear often in our book, and having a common vocabulary is a crucial component in your acquiring information.

The MIDI STANDARD refers to a concrete description of the total communication structure between electronic synthesizers and related devices, such as the Atari ST computer. This includes both the hardware apparatus, cables, and certain properties of the electronic synthesizers. For example, saying that a synthesizer should have the electronics built in to allow the Atari ST to send a message over a cable to turn a note on and off is to discuss the MIDI STANDARD. To say that a keyboard should have 16 channels is to discuss the MIDI STANDARD.

The MIDI LANGUAGE, which will be the main topic of Chapter II involves the nature of the data which is transferred between the Atari ST and the synthesizer. This language is composed of sequences of numbers and can be spoken using any programming language which allows you to access the MIDI interface within the input/output structure of the ST. Virtually all languages we have encountered allow such interaction.

A MIDI IMPLEMENTATION refers to the physical presence of what the MIDI STANDARD call for. For example, one sound quality a note may possess is VIBRATO. VIBRATO is outlined in the MIDI STANDARD. The IMPLEMENTATION in one synthesizer might allow the Atari ST owner to turn the VIBRATO on and off, to set the rate, and depth. Another synthesizer might not allow the changing of the VIBRATO rate and depth. We could say in that instance that the two synthesizers did not have the same IMPLEMENTATION of the MIDI STANDARD.

In a little while we are going to outline some ideas which you may consider when beginning your shopping for a MIDI synthesizer. One thing we will encourage you to do is examine the MIDI IMPLEMENTATION of individual synthesizers to assess which one will best fit your needs.

In summary, the MIDI SPECIFICATION 1.0 lays out the overall structure for the hardware interface and the communication protocol between electronic musical instruments and the Atari ST. In this book we are calling that aspect of the IMPLEMENTATION the MIDI STANDARD. The part of the SPECIFICATION which outlines the nature of the MIDI data and its syntax we call the MIDI LANGUAGE.

Any time there is a standard set up there will always be people who say, "That standard isn't good enough for me!" Or people who say, "People don't need all of the features contained in the standard!" The MIDI originators decided to create a baseline standard and then allow individual manufacturers to add bells and whistles to that standard.

The keyboard synthesizer's trademark is a piano-like keyboard inlaid in a small case. Within the case are a variety of electronic devices used to produce the notes, generate a multitude of instrument simulations and hook into other MIDI-related devices.

In the following paragraphs we will describe some of the more important features of the keyboard synthesizer. For this, we will also need to define a series of both musical and MIDI terms. Understanding the vocabulary at this juncture will allow for easier reading as you continue in the book. Since it is our experience that many people become confused by the differences between, say, MODES and CHANNELS, we have consciously decided to explicitly state these terms' meaning when they are used in key sections in the book. Although this method of organization might well seem redundant to some, we are willing to belabor certain points and risk flak for this repetitive style in an effort to reinforce the learning of key MIDI concepts.

There is one other point to know before we continue. Not all of the features described in the following paragraphs will be operative on every synthesizer on the market. You will need to read your synthesizer manual very carefully in order to know whether your synthesizer has the feature in question. For example, not every synthesizer allows the user to change its VELOCITY (a measure of note loudness). For one of those synthesizers it would appear that sending a VELOCITY message to a synthesizer which doesn't have adjustable velocity does no harm. The message should be ignored.

Now, to begin:

Pressing a key will produce the sound of a NOTE which corresponds to the note's frequency. By that we mean, if you press the middle 'C' key on the keyboard, the middle 'C' note will play. That is simple enough.

Pressing a key is not the only way to play a note on a synthesizer, though. Your Atari ST, after being properly hooked up to your synthesizer, might just as easily send a NOTE ON message through the 5 pin DIN connecting cables. Like electronic magic, the note will play. Sending the NOTE OFF message will, conversely, turn the note off.

NOTE ON and NOTE OFF messages are not the only information that the Atari ST can SEND and RECEIVE. RECEIVE? Yes, the ST can also "mind meld" with the synthesizer to read and remember all that transpires by human initiation on the keyboard.

The INSTRUMENT VOICE means the simulated tone the synthesizer is producing. Some common instruments available on keyboard synthesizers are flute, bass, trumpet, piano, whistle, oboe, etc. For example, you might select the flute INSTRUMENT VOICE tone to be used as you play notes on the synthesizer. Then when you press keys it will produce a sound close to that of a flute.

PROGRAM CHANGE or PROGRAM SELECT refers to the process of changing the instrument voice that the synthesizer is playing. The instrument voices can fall into three catagories: PRESET, INTERNAL, and CARTRIDGE. The PRESET voices are assigned numbers ranging from 1 - 32. These instrument voices are preset at the synthesizer factory and can not be changed by the programmer. The PROGRAM CHANGE numbers 33 - 64 have been reserved for the internal instrument voices. These voices may be created by the synthesizer user (see PROGRAMMING YOUR SYNTHESIZER) and are either remembered by the synthesizer or are transferred to the synthesizer from your Atari ST. PROGRAM CHANGE numbers 65 - 96 are reserved for plug in CARTRIDGES. These special cartridges have the ability to add even more special instrument voices to your already impressive assortment of choices. This change can be accomplished manually or by having the computer, using the appropriate MIDI language code, enable the PROGRAM CHANGE.

The speed at which you press the key down is called the VELOCITY. Generally speaking, the quicker you press the key the more dynamically, or loud, the note will play.

The AFTERTOUCH refers to the amount of pressure you, as the player, put on the key you are pressing. For example, had you previously selected the flute, increasing the AFTERTOUCH would simulate the flute player blowing a greater volume of air as (s)he played.

VIBRATO is the pulsating effect heard when a note is played. The VIBRATO rate, the speed of the pulsation in the sound, and the depth, the intensity of the pulsation, can be controlled.

PORTAMENTO, to the ear, sounds as a musical slur. Say you play the notes middle 'C' and then the 'D' above middle 'C'. You will hear two distinct notes with two distinct frequencies. As you increase the PORTAMENTO time the distinction between the 'C' and 'D' will begin to disappear. The notes will begin to merge together so that the lower 'C' note will gradually increase in frequency and glide to the higher 'D'.

The PITCH BEND or PITCH WHEEL allows the player to glide the frequency of the playing note in either direction with a turn of the PITCH WHEEL. The range of the glide is controlled by the PITCH BEND or PITCH WHEEL setting.

MIDI keyboards have 16 CHANNELS available to play through. Using your MIDI software you might select the bass instrument voice to play through MIDI CHANNEL 1, and the flute instrument voice to play through CHANNEL 2. Some synthesizers will allow you play 16 different instrument voices on the 16 available CHANNELS. The BASIC CHANNEL may defined as a kind of bottom foundation CHANNEL. For example, in the Casio CZ-101 electronic keyboard synthesizer you may set the BASIC CHANNEL to, say, 5. This would mean that the Casio would respond to note on and note off messages on CHANNELS 5, 6, 7, & 8. We refer you to a Casio manual for the reason why this will occur, but its importance cannot be over-emphasized. For example, the Casio CZ 101 can play instrument voices out of a maximum of four CHANNELS. If you hooked up two Casios together and set one's BASIC CHANNEL to 1 and the other's BASIC CHANNEL to 5 you could send 8 voices of music from the Atari ST to your two Casio synthesizers. The CZ-101 with its BASIC CHANNEL set to 1 will play all note on and note off messages on CHANNELS 1 - 4. The CZ-101 with its BASIC CHANNEL set to 5 will play all note on and note off messages on CHANNELS 5 - 8. That way, two CZ-101s could play 8 voices!

The MIDI standard calls for four different keyboard MODES. The MODE that the synthesizer is set to determines how it will interpret and consequently play incoming note data. The MODES are: OMNI ON/POLY, OMNI ON/MONO, OMNI OFF/POLY and OMNI OFF/MONO.

The OMNI/POLY or OMNI MODE sets the electronic keyboard to receive note information on any of the synthesizer's 16 channels. The catch is that all 16 voices are played using a single instrument voice. For example, if you were feeding the synthesizer a four-part fugue, each part would be assigned an individual channel. But, each channel would have the same instrument assigned to it. So you might choose to have four flutes, or four whistlers, or four violins playing the fugue. This MODE might prove useful to someone who for some reason needs a decent sounding piece quickly and is a very beginner with keyboard synthesizers. With a bit more time the beginner can learn how to do much more than the OMNI/POLY MODE will allow. Overall, it seems that the OMNI/POLY MODE isn't too useful.

The OMNI ON/MONO mode will take all the note messages that are coming into the keyboard and put them in just one voice. Simply stated, if you sent 16 voices of note data from your Atari ST, it would come out as a single line of music. This playing of a single note at a time would eliminate the playing of chords. It would sound hollow compared to a full voice implementation. This mode produces sounds similar to how the early model synthesizers sounded as they also allowed only one note to be played at a time. The OMNI ON/ MONO mode does not seem very useful when using one keyboard synthesizer.

The OMNI OFF/ POLY or POLY mode requires that the Atari ST and the keyboard synthesizer be tuned to the same channel for data transfer. If, say, the Atari ST was sending note data on channel 1 and the keyboard synthesizer was set to receive on channel 2, no sound would be played. As with the previous two modes, the OMNI OFF/ POLY mode might prove useful in a multi-MIDI implementation, but for the single MIDI implementation it serves poorly.

Don't get disheartened. We've saved the best for last. The OMNI OFF/ MONO mode is the mode of choice for the single MIDI keyboard implementation. The mode will allow the user to transmit from the computer 16 voices, with one voice appropriately being sent to each channel. EACH CHANNEL MAY HAVE ITS OWN INSTRUMENT VOICE! This means that channel 1 could play a flute, channel 2 a violin, channel 3 a snare drum...etc. It is in the OMNI OFF/ MONO mode that the power of the Atari ST and the keyboard synthesizer come to shine.

We do believe that part of the confusion surrounding the OMNI/MONO mode is its very name. MONO seems to imply, at least to us, that it will play one voice. In fact our initial confusion with the powerful MONO mode came when we fired up our Casio synthesizers. Why? Because if you press the MONO button on our CZ-101 you can play only one note at a time from the keyboard. With the button in the up position you could play chords. So when we tested the first code for the player we did not press the MONO button. All we heard was a single line of music playing. We nearly went to the looney bin trying to figure out what was happening. We then pressed the MONO button out of desperation, and presto, all four voices played with great pride. Pressing the MONO button on the Casio to achieve multi-voice play seemed rather daffy, but so it is, and so it goes.

The distinctions between the MIDI MODE, CHANNEL, INSTRUMENT VOICE and PROGRAM CHANGE can be confusing at times. As we encounter these terms later in the book they will be explained again. Hopefully, the repetition and clarity will bring greater understanding.

As we suggested earlier in this chapter one of the powerful aspects of electronic synthesizers is their ability to simulate the sound of different instruments. The range of synthesizer sounds extends far beyond the number of traditional acoustic instruments.

There are a variety of electronic methods for synthesizing sounds. Some among then have names such as: Analog, Digital Phase Distortion, Digital FM, and cousins and combinations of the previously mentioned three. At this point in the book we could give a brief discourse on each of the mentioned electronic methods of synthesizing sound, but we feel that the discussion would be too tangential for the goals of this book. What we view as more important is: How difficult is it for the synthesizer owner to change the quality of sound that the synthesizer produces?

The electronics of synthetic sound generation allow for a variety of parameters to be adjusted. Altering the numerical values of these parameters will change the sound that the synthesizer is producing. The methods used to change the numbers which describe the synthesizer's sound are most often different for varying brands of synthesizers and are traditionally explained in full detail in the synthesizer's operating manual.

We have reached another point where it is once again beneficial to define a few more terms. It is common practice in magazines and books aligned closely to the music industry to refer to altering the numbers which change the sounds that the synthesizer is producing as "PROGRAMMING THE SYNTHESIZER". As we come from the computer programming community we feel that some readers might be confused by that term.

It is for this reason that when referring to programming the Atari ST, we will specifically state that case. For example, we might say, "If you enter the source code for the program `'sample.c'` into your Atari ST and then use your 'C' compiler and assembler.....". We will adhere to what appears to be an accepted music writing industry convention of referring to "programming the synthesizer" as that process of altering the numbers which change the sound qualities of the notes produced.

If this proves an inconvenience, we apologize, but clarity in transfer of information is one of our primary goals. We would rather add a few extra words than leave the reader more confused than need be. Also, when you encounter the phrase "Programming your Synthesizer" in ***Keyboard*** or ***Keyboards & Computers*** magazines, you will immediately know that you will NOT find a program for your Atari ST.

Collections of the numbers which tell the synthesizer what type of sound to play are generally called PATCHES. The word PATCHES is a throwback to the early days of synthesizers where patches of connecting wires were used to alter the synthesizer's sounds. PATCHES are made up of numbers with the formal names of TONE DATA or SOUND DATA. Restated:

TONE DATA or SOUND DATA refers to the parameters, or number settings which make the synthesizer voice sound the way it does. The Casio CZ-101 allows the programmer and synthesizer player to change a veritable cornucopia of parameters. Some include: Line Select, Ring, Noise, Vibrato Rate, Vibrato Depth, various Wave Forms, Envelope Shape...etc.

We hope that you are beginning to see the richness and fantastic musical opportunities afforded to the owners of the Atari ST and electronic synthesizers. This book will give the musician Atari ST owner and the programming professional and casual user alike a wealth of ideas for fun projects. But, know that as much as is presented here, this book only scratches the surface of the ever-changing universe of electronic music.

1.6 Programming Your Synthesizer

Programming your synthesizer is not programming your Atari ST computer. It is the method used to change the ways the synthesizer sounds are generated.

Traditionally, the buttons, wheels and dials used to change the quality of the synthesizer's sounds are located on the synthesizer's cabinet. Each control device is attached to a part of the electronic circuitry which will change the produced sound in some way. For the purposes of this discussion we will say that a PARAMETER represents an aspect of the synthesizer's sound. The PARAMETER can have many different settings and can be adjusted by one of the control devices on the synthesizer's case. For example, vibrato would be an example of a sound parameter.

Here is a precisely described example of how you might program the Casio CZ-101 to change the vibrato for a voice. The instructions are in English. The parameter of the vibrato we are changing is the vibrato WAVE FORM. If you have a CZ series synthesizer and you are not familiar with programming it yet, now might be a good time to try this little example.

1) Turn on the CZ-101 synthesizer
2) Press the gray #2 button for switching on the preset voice 2 internal sound quality.
3) The vibrato ON/OFF light will go on. Press the gray vibrato button.
4) The LCD text window will display:
   WAVE=1 DELAY=27 RATE=49 DEPTH=49
5) Notice the blinking underscore below the 1 next to WAVE. Press the UP ARROW KEY.
6) Now the WAVE=2.

There you have it! All that just to change the vibrato WAVE FORM.

Not all sound parameters are the same for each electronic synthesizer. For example, the Casio CZ-101 uses the Phase Distortion method of electronic sound synthesis. Returning to the previously mentioned text display in the CZ's LCD window, there are four parameters mentioned in the CZ's description of the vibrato sound. Another brand of synthesizer might call those parameters by different names, only have two of the four parameters, or not have vibrato at all. The features your synthesizer has are all determined by its MIDI IMPLEMENTATION.

There is not a one-to-one relationship between a synthesizer's SOUND DATA (the actual numbers) and the sound that is produced by the synthesizer. In fact, creating a sound can be somewhat tricky. For example, say your synthesizer has two parameters called "A" and "B". "A" is set to 49 and "B" is set to 70. You do the following:

1) Listen to the sound.
2) Change "A" from 49 to 10.
3) Listen to the sound. There is no change in the sound!
4) Change "B" from 70 to 90.
5) Listen to the sound. There IS a change in the sound.
5) Return to "A" and change "A" from 10 back to 49.
6) Listen to the sound. There IS a change in the sound!

The change in "A" did not affect the sound until "B"'s setting was at 90. Another way to describe what we are trying explain is to say that "A" and "B" have a mutually symbiotic relationship. Changing one member in the SOUND DATA's number set may change the relationship of the other SOUND DATA to the sound that the synthesizer produces.

We are not done with our discussion of programming your synthesizer yet! There is a way to have your Atari ST X-ray your synthesizer's SOUND DATA and store the SOUND DATA for a particular voice on a disk file. The type of program that does this can be called a PATCH EDITOR or PATCH LIBRARIAN. PATCH EDITORs can also tell your Atari ST to send the SOUND DATA back to your synthesizer. The SOUND DATA loads to any INTERNAL or RAM CARTRIDGE voice. The reason why some companies call their PATCH related programs LIBRARIANS is that they have the capability of storing literally hundreds of sound PATCHES on disk. The patch files could be called your LIBRARY of SOUNDS.

There are many ways to begin developing your skills as a synthesizer programmer. We suggest that you use two time-honored methods of learning. First, start out with a PRESET VOICE. Get a pencil and paper and begin, in an organized fashion, to change one sound parameter at a time. Listen to the change. Change it a little more. Listen to the sound again. Write your impression of what your change did. Select another sound parameter...and continue the process until you feel that it is time to stop. The second way would be to search out the magazines and books to look for other people's published patches. Program them into your synthesizer. Listen to the results. Follow the first procedure described. Learning to program a synthesizer is a prime example of the maxim: Experimentation is the mother of invention! Enjoy!

1.7 A Guide For Buying Your Synthesizer

Clearly, many people are fascinated with the electronics of sound synthesis. For the purposes of this book, though, we will argue that the end result of how the synthesizer sounds and how easy it is to program (remember... "programming the synthesizer") is more important to most users than the electronics of producing the sounds.

For most applications, if you've got $1,500 to spend you'll have little trouble finding a full-featured keyboard synthesizer for most applications. Since we didn't (and still don't) have $1,500 to spend on a full featured synthesizer, we reasoned that it would make sense to lay out a few points to think about when planning the purchase of your first MIDI keyboard.

The first question we would place on our list would be, "Do you or does anyone in your family play the piano?". If the answer to that question is YES, then it would be wise to bring the person or people along to test out the action of the synthesizer. Keyboards can have different key responses and different numbers of keys making up the keyboard. If there is a choice between two equals, the feel of the keyboard or the number of keys it contains might present a deciding factor. If the answer is NO then you might be able to look at some keyboard synthesizers which have reduced size keyboards.

What modes may be enabled by the synthesizer? Some of the MODES might prove useful in multiple electronic instrument implementations. Is that a consideration for you?

Another matter to consider would be to explore how many instrument voices could be played in the synthesizer's MONO mode. If you remember, the MONO mode allows each voice to be played through an individual channel while having a different instrument selected. The number of channels used for the MONO mode currently ranges from 4 to 16 channels. Will having four individual voices using four individually defined instruments suffice? If not, how many will you need for your application? Six? Eight? It appears that the more channels you wish to have available to you in the MONO mode, the higher the cost.

How is the MIDI STANDARD represented in the synthesizer's MIDI IMPLEMENTATION? Remember that the keyboard synthesizer can receive data from your Atari ST and also transmit data to your Atari ST. The keyboard will receive data if, say, your Atari ST has software which is communicating using the MIDI LANGUAGE and telling it to play a song.

The keyboard synthesizer will be transmitting data to the Atari ST if you are playing a song at the keyboard and wish to have your Atari ST remember all the notes that have been played. Sometimes synthesizers do not have the same MIDI IMPLEMENTATION for the transmitting to the Atari ST and the receiving from the Atari ST. For example, the Akai AX60 keyboard synthesizer can transmit a velocity value, but it cannot receive a velocity value.

The manuals for the synthesizer in question will have a copy of the electronic instrument's MIDI IMPLEMENTATION CHART. This chart is composed of four columns which describe the synthesizer's MIDI IMPLEMENTATION. They are: 1)MIDI FUNCTION, such as velocity, after pressure, pitch bend...etc.; 2) TRANSMIT, whether it transmits information to implement the function in question; 3) RECEIVE, whether it receives information to implement the function in question; 4) REMARKS, contain a brief space for special explanations. It might pay to have a careful look at the MIDI IMPLEMENTATION chart of any synthesizer you are considering buying.

Model MIDI Implementation Chart Date: Version:

Function...		Transmitted	Recognized	Remarks
Basic Channel	Default Channel			
Mode	Default Messages Altered			
Note Number	True Voice			
Velocity	Note ON Note OFF			
After Touch	Key's Ch's			
Pitch Bender				
Control Change				
Prog Change	True #			
System Exclusive				
System Common	: Song Pos : Song Sel : Tune			
System Real Time	: Clock : Commands			
Aux Mes-sages	: Local ON/OFF : All Notes Off : Active Sense : Reset			
Notes				

Mode 1 : OMNI ON, POLY Mode 2 : OMNI ON, MONO X: Yes
Mode 3 : OMNI OFF, POLY Mode 4 : OMNI OFF, MONO O : No

Price, of course, will have a critical impact on which keyboard synthesizer you choose to get. Finding the price/performance ratio which suits your needs best is the trick. Here is a partial list of MIDI keyboard manufacturers: Akai, Casio, Chroma, Korg, Oxford, Roland, Seiko, Siel, Sequential, Suzuki, Wersi and Yamaha. Prices range from a few hundred dollars up to the $1,500 (U.S.) range. In the New York area the best places to shop for the electronic keyboard synthesizers turned out to be the rock/jazz music shops. Although it was mild culture shock for us middle-aged folks to wander amongst the young'uns, it was well worth the effort. The selection of electronic keyboard offerings in these establishments far surpassed those in the standard piano stores we visited. We chose to buy Casio CZ-101 keyboard synthesizers. This model has a reduced-size keyboard and 4 channels operative for instrument voice assignments in the MONO mode. We chose this model because it was the cheapest we could find with the minimum MIDI IMPLEMENTATION needed to write ST MUSIC BOX and our AUTO-MUSIC MIDI PLAYER. Are we happy? Yes and no. The CZ-101 does everything it is billed to do. When we were keying in our demonstration tunes for some computer shows where the ST MUSIC BOX was going to be shown, there were times when I wished the CZ-101 allowed for the playing of 8 instrument voices through 8 channels rather than the 4 permitted by its MIDI IMPLEMENTATION. It would have permitted dramatically more impressive musical arrangements for the demonstration songs. Would the reception of ST MUSIC BOX have been different by the press and distributor's if we had used 8 instrument voice demonstrations rather than 4 instrument voices? Who knows. Our second consideration was the reduced size keyboard. Neither of us have sophisticated keyboard skills (Len is a well-trained jazz guitarist) so we decided that the reduced size keyboard wouldn't matter. But- Len has a twelve year old daughter and Dennis has two children who might like to have a full sized piano keyboard to play on.

In retrospect, I suppose we would have gone for the Casio CZ-1000, with its 49-full sized keys. We would still have forgone the 8 instrument voices as a non-necessary luxury. Good hunting!

1.8 A Guide for buying Your MIDI Software

We have been on both ends of the software industry. We are users. We are software designers and assembly/'C' programmers. We have read reviews and bought products and our products have been reviewed and bought. We know everyone has their opinions, and so do we. Here are our thoughts from a user perspective. In Chapter III we will present our programmer perspective as we begin the massive operation of explaining the logic structure for the 'C' source code presented in this volume.

First we will consider some qualities in programs where the electronic keyboard takes on the role of the receiver and the Atari ST is telling the keyboard what to do.

Before we begin, we need to define a few musical terms for those not familiar with their meanings.

Tempo	the speed at which the song is played
Duration	how long a note is held
Key	here, the foundation or home note
Manuscript	here, written or printed music
Editor	program used to enter musical notes

When determining what software you should first begin by assessing your own needs. For example, do you need to print the manuscript? Do you need to have all the pieces and/or voices displayed at the same time on the printed manuscript? Do you compose for a piano? Do you compose classical quartets? Are you a popular song composer and need more than a rudimentary text inclusion function in order to print your newest hits?

What do you need in the editor? What is the method of note entry? How is the editor structured? Does it use the mouse? Are the functions like INSERT MEASURE, DELETE MEASURE?

Once you have entered the notes into the music editor how will the notes be played? What are the MIDI functions which can be written to? Can you control the instrument voice per line of music? Can you have different instruments per voice? When do you select the instrument? Per song? Per measure? Per note? When do you select the song's tempo? Per song? Per measure? Per note?

Once again we need to define another MIDI world buzz word.

REAL TIME CAPTURE is the ability of an Atari ST computer program to receive the keyboard information transmitted FROM the MIDI electronic keyboard and store it in memory. Once the song information is in memory then you will be able to alter it any way you want. REAL TIME CAPTURE might be an important feature if you have keyboard skills where you would want to play a song, rather than enter the notes with an editor and let the Atari ST play the song.

What are you going to use the MIDI software for? If you don't play the piano, REAL TIME CAPTURE may not be a useful feature to have. What type of manuscript print program suits your needs best? If you're not too familiar with entering music into an editor, then getting a smart one with a syntax check would probably be smart. If you sing, getting a program which would allow you to change the song's keys would be a critical feature in adjusting the music to your range.

We've presented some ideas for you to chew on. Our advice is to look carefully when examining the software you are considering for purchase. Ask to see the programs demonstrated at your local computer store. Talk to friends who are already using MIDI software. If you ask yourself some of the questions we raised in the preceding paragraphs and then answer them, you will be well on the way to finding the MIDI program for you.

One other note: we do a great deal of writing. Some of the writing is either 'C' or assembly source code. We also write non-fiction and have dabbled in fiction. We use three text editors on the Atari ST computers and two text editors on our 130XE computers. We use so many because each editor is slightly different from the next one, and each one fills a different need based on the type of writing we are doing. What we are saying is that if you are serious about entering the splendor of MIDI-land you just might want to have a few Atari ST MIDI programs to work with. Just a thought.

1.9 Chapter 1 Summary

In Chapter I we explained that the MIDI SPECIFICATION 1.0 is a mutually-agreed-upon standard of hardware and software principles, allowing electronic musical instruments and computers to speak with each other. For convenience we broke the MIDI SPECIFICATION 1.0 into two distinct categories. The first we called the MIDI STANDARD. This category included all the interface hardware and a base line standard of electronic synthesizer functions. The other category derived from the MIDI SPECIFICATION 1.0 is the MIDI LANGUAGE. This is the language which you program your Atari ST to use in communicating to the electronic synthesizer. MIDI LANGUAGE is the primary subject of Chapter II.

The functions of the electronic synthesizer allow your Atari ST to send messages through the interconnecting 5 pin DIN cables to your synthesizer which will command it to play songs with amazing versatility. Also, the Atari ST can function as a receiver, remembering all the action on the synthesizer keyboard. It is the software running the Atari ST which determines how it will communicate with the electronic synthesizer.

Programming your synthesizer means changing the sound quality produced by one or more of the synthesizer's instrument voices. Programming the Atari ST means telling the ST how you wish it to communicate with the synthesizer. In this book, programs for the Atari ST will be written in the 'C' programming language. There are now a goodly number of fine 'C' compilers available for the Atari ST and many 'C' programming instruction texts on bookstore shelves.

Chapter II

The MIDI LANGUAGE

The MIDI LANGUAGE

The Atari ST is connected to your keyboard synthesizer by the 5 pin DIN cables plugged at one end into the synthesizer's MIDI ports and at the other end into the ST's MIDI ports. The connecting cable will carry electrical impulses between the synthesizer and the ST. These electrical impulses will represent numerical values.

The MIDI LANGUAGE, in a sense, can be considered a mathematical language where the basic communication elements are clusters of numbers. In this chapter we will define what the clusters of numbers mean. The process of getting your Atari ST to bring the synthesizer to life is to have the ST send the appropriate clusters of numbers through the connecting MIDI cable to the synthesizer.

2.1 BITS & BYTES

For purposes of enriching this chapter for those not familiar with the world of BITS & BYTES, we have decided to include the information presented in this section. If you are familiar with these sacred computer concepts then GOTO section 2.2.

The MIDI STANDARD calls for the special number clusters of the MIDI LANGUAGE to be held in bytes. A BYTE may be thought of as a container which can hold numbers with values ranging from 0 to 255. When you say you are sending a BYTE of information, all you are saying is that you are sending a number between 0 and 255.

A BYTE is composed of 8 BITS. Following the previous analogy of a BYTE being a container, the 8 BITS could be conceived of as an arrangement of 8 compartments in the container. Each BIT compartment will have a specific location with number identifiers ranging from 0 to 7.

Still with us? Each of the 8 BIT location compartments can have either a 0 or 1 in them. The numerical value of the BYTE is determined by the specific BIT compartments which hold 1s. Table 2.A is a CONVERSION TABLE showing the relationship between the BIT COMPARTMENT LOCATION and DECIMAL CONVERSION number.

Table 2.A

COMPARTMENT LOCATION	DECIMAL CONVERSION IF BIT COMPARTMENT HOLDS a 1
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

If ANY BIT COMPARTMENT holds a 0 the conversion number will ALWAYS be 0.

The process of converting a number from BINARY to DECIMAL is really quite simple. For purposes of this book we will explain how to convert BINARY numbers to DECIMAL numbers starting with BIT COMPARTMENT LOCATION 7. Here is the process:

<u>BINARY to DECIMAL BYTE</u>

1) Start with a variable we'll call BYTE set to 0.

2) Examine BIT COMPARTMENT 7 of BINARY NUMBER. If it holds a 1 then add the DECIMAL CONVERSION number for BIT LOCATION 7 (128) to BYTE (see Table 2.A if you don't understand where the 128 comes from).

3) Examine BIT COMPARTMENT 6 of BINARY NUMBER. If it holds a 1 then add the DECIMAL CONVERSION number for BIT LOCATION 6 (64) to BYTE.

.
.
.
.

9) Examine BIT COMPARTMENT 0 of BINARY NUMBER. If it holds a 1 then add the DECIMAL CONVERSION number for BIT LOCATION 0 (1) to BYTE.

10) Variable BYTE holds the DECIMAL equivalent of the BINARY number.

The numerical value of a BYTE is composed of the addition of all the BIT DECIMAL CONVERSION VALUES. If all the BIT compartments hold a value of 1, then all the DECIMAL CONVERSION VALUES would be summed to get the BYTE value. You would have: 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255. If every one of the 8 BIT COMPARTMENTS held a 1, and all the DECIMAL CONVERSIONS were used for the BYTE SUMMATION, the maximum value of the total would be 255. That is why a BYTE (composed of 8 BIT COMPARTMENTS) can hold a number no higher than 255.

Let's try to figure out the decimal value of a BYTE with the following BIT COMPARTMENT arrangement. See Table 2.B.

See that the COMPARTMENT LOCATIONS range between 0 and 7. The COMPARTMENT VALUES either hold a 0 or 1. Note that if the COMPARTMENT VALUE is 0 the CONVERSION VALUE is ALWAYS 0. If the BIT COMPARTMENT VALUE holds a 1 the the DECIMAL CONVERSION value is determined by the BIT COMPARTMENT LOCATION.

In Table 2.B we see that BIT COMPARTMENT LOCATION 0 holds a 0. Following the previously mentioned rule: THE DECIMAL CONVERSION is ALWAYS 0 if the BIT COMPARTMENT holds a 0. In column 3, CONVERSION SUMMATION, the value is 0 on the COMPARTMENT LOCATION 0 row.

Table 2.B

Compartment Location	COMPARTMENT VALUE	CONVERSION SUMMATION
0	0	0
1	0	0
2	1	4
3	1	8
4	0	0
5	1	32
6	0	0
7	1	128
	BYTE VALUE =	172

In Table 2.B we see that COMPARTMENT LOCATION 2 is the first COMPARTMENT holding a 1. Look at Table 2.A now. The CONVERSION VALUE for for COMPARTMENT LOCATION 2 is 4. Returning to Table 2.B we then put that 4 into the CONVERSION SUMMATION column.

Why is there a 32 in Table 2.B's CONVERSION SUMMATION column? If your answer is "because BIT COMPARTMENT LOCATION 5 holds a 1" you are CORRECT!

Traditionally the arrangement of the BIT COMPARTMENTS are presented horizontally and not vertically. For the remainder of this book we will follow the following convention:

```
COMPARTMENT
LOCATION        7  6  5  4  3  2  1  0
                _  _  _  _  _  _  _  _
COMPARTMENT
VALUE           1  1  1  1  1  1  1  1 = 255
```

The BIT COMPARTMENT LOCATION on the RIGHT is 0. It will always be presented as 0. As you move left the COMPARTMENT LOCATIONS will increment. The = 255 represents the summation of all the DECIMAL CONVERSION VALUES.

Let's try another example. Using Table 2.A can you understand the following BIT to BYTE conversion?

```
 BIT #
7  6  5  4  3  2  1  0
_  _  _  _  _  _  _  _
0  0  0  0  0  0  1  1 = 3
```

It will not be necessary to fully understand the process of BIT to BYTE conversion to understand the MIDI LANGUAGE. For those musicians who didn't have any knowledge of the hows and whys of bits and bytes we have presented this very brief explanation to enrich our discussion of the MIDI LANGUAGE.

One other note: for the remainder of this chapter we will use DECIMAL numbers to explain the MIDI language, as it is most commonly understood. In later chapters containing the 'C' source code we will use both hexadecimal and decimal notation. It is not within the scope of this Atari ST and MIDI book to explain all of the ins and out of binary to decimal to hexidecimal to octal conversions, and we refer interested readers to basic 'C' language programming texts.

2.2 Data FORMAT

It is germane at this time to discuss the types of messages which can transpire between MIDI electronic instruments. For purposes of this discussion we will say that there are two main catagories of MESSAGE TYPES. They are called: CHANNEL and SYSTEM.

The CHANNEL MESSAGE TYPE may be further divided by function into two more categories. They are called: VOICE and MODE.

The VOICE MESSAGE TYPE contains information such as "Turn a note on in channel ..", "Turn a note off in channel...", etc. The full list of VOICE MESSAGES will be presented shortly.

The MODE MESSAGE TYPE contains information which SELECTS the BASIC CHANNEL (at this time thinking of it as a foundation CHANNEL will suffice) synthesizer MODE (see previous discussion on MODES if confused).

Within the category of SYSTEM MESSAGE TYPES there are three sub-categories. They are called: COMMON, REAL-TIME, and EXCLUSIVE.

SYSTEM's COMMON MESSAGE TYPES may be understood and implemented by all MIDI synthesizer instruments hooked up to your Atari ST. They might include turning notes on, turning notes off, etc.

REAL-TIME MESSAGE TYPES are used for MIDI electronic instrument hook-ups more complicated than, say, hooking your Atari ST to an electronic keyboard synthesizer. An example of where a REAL-TIME MESSAGE TYPE would be necessary would be if you were going to hook up a MIDI electronic DRUM MACHINE and a MIDI electronic keyboard synthesizer to your Atari ST. Your electronic DRUM MACHINE's timing must be synchronized to the playing of your electronic MIDI keyboard. If the electronic DRUM MACHINE is not synchronized with the electronic keyboard then you would have some unwanted creative cacophony. REAL-TIME messages would allow you to synchronize the timing between your Atari ST, MIDI electronic keyboard synthesizer, and MIDI electronic DRUM MACHINE.

SYSTEM's EXCLUSIVE messages are specifically designed to be read and interpreted by your specific brand of electronic keyboard synthesizer. For example, PATCHES (see discussion on PATCHES in Chapter 1) may be sent from your Atari ST to your specific brand of synthesizer using a SYSTEM's EXCLUSIVE call. Each MIDI manufacturer has what is called a MIDI ID code. This manufacturer ID code must be packaged in your SYSTEM's EXCLUSIVE message in order for your electronic keyboard synthesizer to know that the following SYSTEM's EXCLUSIVE is meant especially for its ears. An example of programming your Atari ST to send a SYSTEM's EXCLUSIVE message will be given in Chapter III.

We must now return to defining terms. We will call the numbers that are sent from your Atari ST to your keyboard synthesizer MIDI MESSAGES. These message numbers will be called DATA. There are two MIDI DATA TYPES.They are called: STATUS BYTES and DATA BYTES.

The STATUS BYTE tells the electronic synthesizer, "Use the following information in this way!". A specific use of the STATUS BYTE would be to send the following message:

1) Turn a note on using the instrument voice assigned to channel 1.

Often after a STATUS BYTE is sent from your Atari ST to your MIDI synthesizer, one or two DATA BYTES are sent. The DATA BYTE defines the content of the message. Continuing the specific use of the above mentioned STATUS BYTE we will add the content messages of two DATA BYTES.

2) The note to be turned on is middle 'C'.
3) Turn the note on with a velocity of 64.

You can see that the STATUS BYTE of a NOTE ON message is composed of a three byte sequence. The NOTE ON STATUS BYTE, the NOTE and the VELOCITY can be represented by three numbers. If these three specific numbers are sent from your Atari ST's MIDI port to the keyboard synthesizer the note MIDDLE 'C' will play.

What follows will be the standard MIDI 1.0 Detailed Specifications and consequent tables. There are certain features of the Specification which are not relevant to the demonstration programs included in this book or appropriate for the beginner with MIDI. We have decided to gloss over these aspects of the Specification with the purpose of deepening the explanations of more relevant topics.

Table 2.C

SUMMARY OF STATUS BYTES

STATUS	# of DATA BYTES	DESCRIPTION
Channel Voice Messages		
bit # 76543210		
1000nnnn	2	Note Off event
1001nnnn	2	Note on event(vel.=0:note off)
1010nnnn	2	Poly key pressure/after touch
1011nnnn	2	Control change
1100nnnn	1	Program change
1101nnnn	1	Channel pressure/after touch
1110nnnn	2	Pitch bend change
Channel Mode Messages		
bit # 76543210		
1011nnnn	2	Selects channel mode
System Messages		
bit # 76543210		
1111000	*****	System Exclusive
11110sss	0 to 2	System Common
11111ttt	0	System Real Time

We'll begin our explanation by exploring the meaning of voice messages. Row 1 reads:

1000nnnn 2 Note off event

The "1000nnnn" is a binary representation (see 2.1 if you're confused by the binary representation) of the STATUS BYTE. A characteristic of a STATUS BYTE is that it will always be EQUAL TO or GREATER THAN 128. The "nnnn" part of the binary representation refers to a channel number. Channels are numbered from 1 to 16 on your synthesizer, but internally you need to subtract 1 from the standard channel listing when calculating your "nnnn". The binary representation for the NOTE OFF EVENT has a 1 in the BIT COMPARTMENT LOCATION 7. By glancing at Table 2.A you can see that the decimal equivalent for a note off event is 128 + the value of the decimal equivalent of nnnn - 1. Fortunately, the 'C' programming language has a very simple method of expressing the mathematical relationship described in the last sentence. The STATUS BYTE required to turn off a note is equal to:

128 + (channel - 1)

That's not too bad. The decimal number following the binary representation of the STATUS BYTE tells how many DATA BYTES follow the STATUS BYTE. Table 2.D is an abridged version of Table 2.C with the binary representations of the STATUS BYTES converted into decimal form.

Table 2.D

STATUS	COMMENTS
128 + (channel # - 1)	Note off
144 + (channel # - 1)	Note on
160 + (channel # - 1)	Poly key press.
176 + (channel # - 1)	Control change
192 + (channel # - 1)	Program change
208 + (channel # - 1)	Chan. press.
224 + (channel # - 1)	Pitch bend

The SYSTEM EXCLUSIVE call is decimal 240 followed by other DATA BYTES. An example 'C' source code for programming the Atari ST for a SYSTEM EXCLUSIVE call for the Casio CZ-101 will be given in Chapter III. Musical NOTES are passed from the Atari ST to your synthesizer in the form of DATA BYTES which are composed of numbers ranging from 0 to 127. The relationship between the actual musical notes and the number scale is represented in Table 2.E

```
                         Table 2.E

 Note 60 = middle C of keyboard

 0  12  24  36  48  60  72  84  96  108  120  127

         C   C   C   C   C   C   C    C
         |—————————— piano range ———————|
```

The Controllers are used to control a variety of electronic synthesizer operations. Some of these operations are specific to individual electronic synthesizers while others are standards agreed upon by manufacturers. Table 2.F reflects the mutually accepted standard Controller numbers.

Table 2.F

STANDARD CONTROLLER VALUES

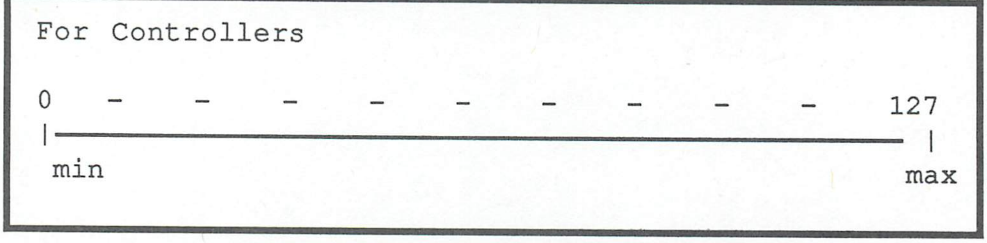
CONTROL NUMBER	CONTROL FUNCTION
0	Undefined
1	Modulation wheel or lever
2	Breath Controller
3	Undefined
4	Foot Controller
5	Portamento time
6	Data Entry
7	Main Volume
8 to 31	Undefined
32 to 63	LSB for values 0 to 31
64	Damper pedal (sustain)
65	Portamento
66	Sustenuto
67	Soft pedal
68 to 95	Undefined
96	Data increment
97	Data decrement
98 to 121	Undefined

Table 2.G

cccccccc	Description
0	Continuous Controller 0 MSB
1	Continuous Controller 1 MSB(MOD BENDER)
2	Continuous Controller 2 MSB
3	Continuous Controller 3 MSB
4-31	Controller 4-31 MSB
32	Continuous Controller 0 LSB
33	Continuous Controller 1 LSB(MOD BENDER)
34	Continuous Controller 2 LSB
35	Continuous Controller 3 LSB
36-63	Continuous Controllers 4-31 LSB
64-95	Switches(ON/OFF)
96-121	Undefined
122-127	Reserved for Channel Mode messages

Controllers can receive values ranging from 0 to 127 as demonstrated by Table 2.H.

Table 2.H



Switches are read as ON/OFF toggles with the value of 0 meaning OFF and the value of 127 meaning ON.

Table 2.I

For Switches

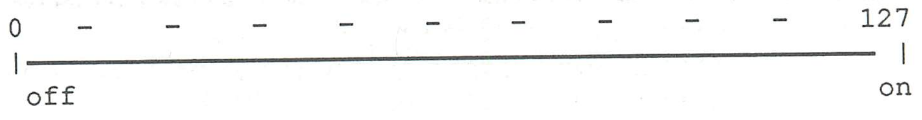


Table 2.J outlines the functions of the CHANNEL MODE MESSAGES.

Numbers 123 to 127 function in such a way as to turn ALL NOTES OFF. These numbered messages should not be used in lieu of regular NOTE OFF commands. Also these ALL NOTE OFF functions will turn off all voices controlled by the assigned BASIC CHANNEL. One other point concerning CHANNEL MODE MESSAGES is that the third "MONO" byte designates the number of channels in which monophonic voice messages are to be sent. The variable "M" is a number between 1 and 16. The channels in operation will be the current BASIC CHANNEL (which = N) through channel (N + M - 1), with 16 as its maximum. Where the special case of M = 0 appears, this causes the synthesizer to assign all its voices to one per channel, ranging from the BASIC CHANNEL N through 16.

Table 2.J

STATUS	DATA BYTES	DESCRIPTION
1011nnn	0ccccccc	Mode Messages
	0vvvvvvv	ccccccc=122:Local Control
		vvvvvvv=0:Loc. Control Off
		vvvvvvv=127:Loc. Control On
		ccccccc=123:All Notes Off
		vvvvvvv=0
		ccccccc=124:ONMI Off(notes off)
		vvvvvvv=0
		ccccccc=125:OMNI ON(notes off)
		vvvvvvv=0
		cccccc=126:MONO ON(Poly OFF) (ALL NOTES OFF)
		vvvvvvv=M, where M = # channels
		vvvvvvv=0, the number of channels equals the number of voices in the receiver
		ccccccc=127:POLY ON(MONO OFF)
		vvvvvvv=0:ALL NOTES OFF

Table 2.K holds the information for SYSTEM COMMON MESSAGES. The SONG POINTER message, SONG SELECT message and TUNE REQUEST message are used for advanced MIDI applications. In the following paragraphs we will simply define their function as per the MIDI 1.0 SPECIFICATION document.

Inside your electronic MIDI keyboard there is an internal MIDI CLOCK. The value of this clock is normally set to 0 as the START switch is pressed. The SONG POINTER holds the MIDI BEATS. A MIDI BEAT is equal to six MIDI CLOCK CYCLES. The value of the SONG POINTER is returned to 0 if the STOP button is pressed.

The SONG SELECT message specifies which song or sequence of songs is to be played upon receipt of a START (Real TIME MESSAGE).

The TUNE REQUEST is used with ANALOG SYNTHESIZERS to request them to tune their oscillators.

The EOX, or "End of System Exclusive" message is used to tell the electronic synthesizer that the SYSTEM EXCLUSIVE call has ended and the bytes subsequently following the EOX message can be handled in a MIDI standard (across instrument) fashion.

Table 2.K

SYSTEM COMMON MESSAGES

STATUS	DATA BYTES	Description
11110001		Undefined
11110010		Song Pointer Position
	0lllllll	lllllll(least significant)
	0hhhhhhh	hhhhhhh(most significant)
11110011	0sssssss	Song Select
		sssssss:Song #
11110100		Undefined
11110101		Undefined
11110110	none	Tune Request
11110111	none	EOX:"End of SYS EXC" flag

SYSTEM REAL TIME MESSAGES are used for advanced MIDI applications. Once again, we will define the messages as they are outlined in the MIDI 1.0 SPECIFICATION DOCUMENT, but will omit detailed explanations as these messages are not used with our demonstration programs.

SYSTEM REAL TIME MESSAGES are used for synchronizing all of the system in real time. These one- or two-byte messages may be sent to the MIDI synthesizer at any time.

The TIMING CLOCK synchronizes the system, with approximately 24 clocks per quarter note.

The START (from the beginning of the song) byte is immediately sent when the master PLAY switch is pressed (on, say, the sequencer or the rhythm unit).

The CONTINUE MESSAGE will tell the sequence to begin at the beginning of the next clock.

The STOP MESSAGE will stop the playing sequence.

Use of the ACTIVE SENSING MESSAGE is optional for all MIDI instruments. This is a "dummy" status byte which is sent every 300 ms., whenever there is no other activity taking place. The electronic MIDI instrument will operate normally if it never receives the ACTIVE SENSING MESSAGE. If the 300 ms. time period passes with no MIDI activity then the ACTIVE SENSING MESSAGE will be sent to the electronic MIDI instrument. The receipt of the ACTIVE SENSING MESSAGE will turn off all the instrument's voices and return the electronic MIDI instrument to normal operation.

The SYSTEM RESET MESSAGE will initialize the electronic MIDI instrument to the condition right after the main electric power to the instrument had been turned on. The SYSTEM RESET should be used very sparingly as it will wipe out all of the new MIDI related information which you have transmitted to the electronic MIDI keyboard.

Table 2.L

SYSTEM REAL TIME MESSAGES

STATUS	DATA BYTES	Description
11111000		Timing Clock
11111001		Undefined
11111010		Start
11111011		Continue
11111100		Stop
11111101		Undefined
11111110		Active Sensing
11111111		System Reset

The SYSTEM EXCLUSIVE MESSAGES are used to tell the electronic synthesizer to perform an operation which is SPECIFIC to that individual synthesizer. A SYSTEM EXCLUSIVE CALL for, say, the Casio CZ-101, will NOT work with a ROLAND synthesizer.

The SYSTEM EXCLUSIVE MESSAGE may be used to change the SOUND DATA for an instrument voice. For example, if you had programmed internal voice 1 to sound like a bamboo flute, you would need to use the SYSTEM EXCLUSIVE MESSAGE to change the sound of the bamboo flute to an electronic guitar.

The specific byte sequences for SYSTEM EXCLUSIVE MESSAGES is published by the individual electronic MIDI instrument manufacturer. You would need to write the manufacturer of your electronic MIDI instrument and request the SYSTEM EXCLUSIVE information in order to obtain it.

Table 2.M starts with the BULK DUMP. The BULK DUMP ETC. message tells the MIDI keyboard that a MIDI EXCLUSIVE byte stream will be forthcoming.

The identification number refers to the specific ID the manufacturer received when they agreed to manufacture their products conforming to the MIDI STANDARD.

The number of bytes sent will vary according to the nature of the EXCLUSIVE call.

The EOX MESSAGE tells the MIDI synthesizer that there will be no more instrument specific information coming to it and it should expect MIDI (cross instrument brand) information in the future.

Table 2.M

STATUS	DATA BYTES	DESCRIPTION
11110000		BULK DUMP ETC.
	0iiiiiii	ID number (can be 1 to 127)
	(0*******)	The number of bytes sent here will be determined by the nature of the system exclusive call. Note that BIT COMPARTMENT 7 will always contain a 0 for the BULK DATA dumped.
	.	
	.	
	.	
	.	
	.	
	11110111	EOX: "End of System Exclusive"

Table 2.N shows the late 1985 list of SYSTEM EXCLUSIVE MANUFACTURER'S ID NUMBERS.

Table 2.N

SYSTEM EXCLUSIVE MANUFACTURER ID NUMBERS

NUMBER	MANUFACTURER		
1	Sequential	32	Bon Tempi (Italy)
2	Big Briar	33	SIEL (Italy)
3	Octave-Platuea	34	IRCAM (France)
4	Moog	35	Synthax (UK)
5	Passport Designs	36	Hohner (Germany)
6	Lexicon	37	Crumar (Italy)
7	Kurzweil	38	Solton
8	Fender	39	Jellinghaus MS
9	Gulbransen	40	CTS
10	Delta Lab Research	41	PPG
11	Sound Compositions Systems	42	Elka
12	General Electro Music		
13	Techmar	64	Kawai
14	Matthews Research & Development	65	Roland
15	Ensonique	66	Korg
6	Oberheim	67	Yamaha
17	PAiA Electronics	68	Casio
18	Simmons Group Centre (UK)	69	Akai
19	Gentle Electric		
20	Fairlight (Australia)		
21	JL Cooper		
22	Lowery		
23	Linn		
24	Emu Systems		

In summary, Chapter II concentrated on explaining the MIDI LANGUAGE. The MIDI LANGUAGE'S vocabulary is composed of numbers ranging from 0 to 255. The MIDI interface in the Atari ST empowers the Atari ST programmer with the ability to send the MIDI vocabulary from the ST to an electronic keyboard synthesizer. This situation allows the Atari ST to play songs using a veritable cornucopia of effects permitted by the electronic keyboard synthesizer. In one sense, the Atari ST's internal MIDI interface has permitted it to be transformed into a simulated human's hands dancing about the keyboard.

Chapter III

Programming Your Atari ST

Programming Your Atari ST

Finally we have arrived at the chapter where we will begin our explanation of the coding techniques we use to speak the MIDI language. But first, for the uninitiated, it would be wise to discuss hexadecimal notation. If you are familiar with hexadecimal notation, then GOTO 3.2.

3.1 Hexadecimal Notation

Earlier in the book we discussed how decimal numbers may be represented using binary notation. This was an important bit of information to learn because the MIDI SPECIFICATION DOCUMENT 1.0 extensively uses binary representation. Understanding hexadecimal notation is important because much of our code uses hexadecimal notation and it will definately be clearer to you if you are conversant with hex.

Hexadecimal notation is not the only notation you might need on the Atari. Why? The answer lies in the General Instrument sound chip manual which uses OCTAL notation! Fortunately, we converted much of the OCTAL numbers contained in the General Instrument's sound chip manual into hexadecimal. The information is contained in the `'mdrive.c'` file printed in Chapter 4.

What follows is Table 3.a which will show the relationship between decimal numbers 0 to 16 and their hexadecimal equivalent.

Table 3.a

Decimal	Hexadecimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f
16	10

It is standard in the 'C' programming language to indicate that a number is hexadecimal by having an "0x" precede it. For example, the number 0x0b would be 0b hexadecimal, and referring to Table 3.a we see that 0b is equal to decimal 11. For the purposes of this book, we will describe the conversion procedure for hexadecimal numbers of up to three digits.

Table 3.b Hexadecimal to Decimal Conversion

Hexadecimal Number

0x02b8

Digit	Operation		Result
0	0 x 4096	=	+000
2	2 x 256	=	+512
b	11 x 1	=	+176
8	8 x 1	=	+008

			696

Hexadecimal to Decimal

0x02b8 = 696

We can see from Table 3.b that the conversion process follows a logical format. Starting with the RIGHT digit, add it to 16 times the next left digit, and add it to 16 x 16 times the next left digit, and add it to 16 x 16 x 16 times the next left digit. Phew.

Fortunately, the time-consuming conversion process may be bypassed by a shortcut which we use all the time. We own Sharp EL-515 scientific calculators. They have very useful binary, octal, hexadecimal and decimal conversion routines built in. The Sharp EL-515 sells for approximately $20 U.S. and is routinely sold by electronic discount stores. If you ever program in any language this calculator will save you a ton of time. If you're doing any work involving the redefining of characters or reading charts such as the General Instrument octal representation of the Atari ST sound chip's parameters it will prove an indispensable time saver.

3.2 The Extended BIOS

There are two convenient ways to write to the MIDI ports on the Atari ST computer using the extended BIOS call 'midiws' and BIOS call 'bconout'. We use the BIOS call 'bconout'. The reasons are simple.We use 'bconout' for many different functions in our programs and it has always performed flawlessly. We subscribe to the philosophy, "If something ain't broken and be workin' fine, DON'T FIX IT!" Enough said.

The BIOS call bconout may be called from 'C' in the following fashion:

```
    Bconout ( device, number )
```

Device and number would both be declared as integers. The integer must be a 16 bit number. Please check your 'C' compiler to see how many bits are assigned to the int declaration. For example, the current version of Lattice 'C' assigns 32 bits to the 'int' declaration and the programmer needs to substitute a 'short' instead of an 'int' in the declaration. Bconout is a very useful call because it does not limit you to sending information to the MIDI. Table 3.c describes bconout's output parameters.

Table 3.c Bconout Parameter Device

Device Number	Output	Device
0	PRT	Centronics Interface
1	AUX	RS 232 Interface
2	CON	Screen
3	MIDI	MIDI Interface
4	IKBD	Intelligent Keyboard

If you wish to send the number 128 through your MIDI port to your MIDI synthesizer, the process is a breeze in 'C'. The following few lines will do the trick:

```
/*
**   declare integer variables
*/

     int  number, device;


/*
**   any number between -32767 and +32767
*/

     number = 128;


/*
**   3 = MIDI...see Table 3.c
*/
     Bconout( device, number );
```

The call 'Bconout(device, number);' will send 128 decimal out the MIDI OUT port.

If you wish to receive information from your electronic MIDI keyboard you would use the `Bconin` BIOS function. The device numbers match those presented in Table 3.c.

```
/*
**   declare two integer variables
*/


     int  device, number;


/*
**   set device = 3 ... MIDI
*/


     device = 3;
```

```
/*
**   the value returned from the MIDI
**   will be placed in number
*/

    number = Bconin( device );
```

The program that follows will demonstrate a call to a Casio CZ series synthesizer which will play a chromatic scale of notes using sixteen different instrument voices. If you remember, an octave is a distance between two pitches composed of 12 gradations of sound called half-tones. An octave is composed of 12 half-tones. `'scaler.c'`'s play will range through 4 octaves or 48 half-tones up and down. This call will demonstrate the use of both the `Bconout` and the `Bconin` BIOS function.

3.3 `scaler.c` source file

Section 3.3 is devoted to writing note data to the to the electronic keyboard synthesizer. This short program has many instructive routines which you may apply to your own programs.

```
/*
** scaler.c
**
** This program will play a 4 octave
** chromatic scale using 16 different
** instrument voices.
**
** This version was compiled using
** Megamax 'C'.  If you have
** Lattice 'C' you will need
** to change all the 'ints' to 'shorts'.
*/

/*
** include definition files here
*/

#include  <osbind.h>
#include  <stdio.h>

/*
** global variables and arrays
** required for gem initialization
*/

int intin[ 256 ], intout[ 256 ], ptsin[ 256 ];
int ptsout[ 256 ], contrl[ 12 ];
int handle, dummy;

/*
** initialize gem
** 1) prepare intin array
** 2) get handle using 'graf_handle...'
** 3) open virtual workstation
*/
```

```
init_gem()

int i;
    for( i = 0; i < 10; i++ )
        intin[ i ] = 1;
    intin[ 10 ] = 2;
    handle = graf_handle( &dummy, &dummy, &dummy, &dummy );
    v_opnvwk( intin, &handle, intout );
}

/*
** Short delay
** There are two nested loops.
** Note the 'd2 = d2' statement.  We
** included this statement because Megamax wouldn't
** compile without it.  This
** delay could easily be modified
** to allow you to control the length
** by passing variables which
** would replace the loop
** terminating conditions of
** 1000 & 10.  Do you see how?
*/

delay()
{
int d1, d2;
    for( d1 = 0; d1 < 1000; d1++ ) {
        for( d2 = 0; d2 < 10; d2++ )
            d2 = d2;
        }
}


/*
** This function will turn a MIDI
** note on.  Using the MIDI
** LANGUAGE and information
** described earlier we see
** that the following sequence of
** bytes must be sent to the MIDI
** in order to turn a note on.
**
```

```
** 1) (128 + 16 + 0 ) -> status for note
**    on in channel 0
** 2) data byte containing
**    note number (36 - 96)
** 3) velocity (0 to 128)
** The Bconout bios call is
** used to send data to the
** MIDI. The '3' parameter is
** the device number for MIDI.
** The 'note_on' function may
** be transported to your own
** programs with ease.
*/

note_on( channel, n_number, velocity )
int channel, n_number, velocity;
{
   Bconout( 3, (128+16+channel) );
   Bconout( 3, n_number );
   Bconout( 3, velocity );
}


/*
** This function will turn a MIDI
** note of.  The following sequence of
** bytes must be sent to the MIDI
** in order to turn a note off.
**
** 1) (128 + 0 ) -> status for note
**    of in channel 0
** 2) data byte containing
**    note number (36 - 96)
** 3) velocity (0 to 128)
** As with the 'note_on' function
** 'note_off' may be easily
** used in your own programs.
*/

note_off( o_channel, o_n_number, o_velocity )
int o_channel, o_n_number, o_velocity;
{
   Bconout( 3, (128+o_channel) );
```

```
    Bconout( 3, o_n_number );
    Bconout( 3, o_velocity );
}


/*
**  This is the main code
**  The first function called is
**  'appl_init()'.  It is
**  used to get an application
**   id number.  We include it
**  here because our programs
**  have no problems initializing
**  gem.  Although gem, once you
**  get your feet wet with the
**  bindings, is a real time
**  saver.  One discomfort we
**  experience using gem, is that
**  we are using code other
**  programmers have written.
**  Remember to include the
**  'appl_init()' call before
**  you open the virtual workstation.
**  When we forgot, we were reminded
**  of our indiscretion with a
**  'shroom display!
*/


main()
{
int    c;
int program, note_val;

/*
**  GEM initialization
*/

    appl_init();
    init_gem();  /* initialize gem */

    v_clrwk( handle );    /* clear screen */
```

```
/*
** begin introduction
*/

   printf("MIDI Chromatic Scale Demonstration \n");
   printf("by Len Dorfman and Dennis Young \n\n");
   printf("Press MONO button on Casio CZ synthesizer\n");
   printf("and then RETURN key to begin. \n ");

   c = getchar();   /* megamax waits for return to exit
                       'getchar()', Alycon needs a Control z */

/*
** This outside 'for' loop changes the
** instrument voice using the MIDI
** LANGUAGE 'PROGRAM CHANGE' message.
** The status byte for the program
** change has bits 7 & 6 on.  Bits
** 0 - 3 are used for the channel
** number.
*/

   for( program = 16; program < 32; program++ )   {
   printf("\nChromatic scale using Instrument voice # %d.
\n", program);
      Bconout( 3, (128+64+0) );
      Bconout( 3, program );

/*
** These nested loops will take all the
** note values ranging from 36 to 95 and
** play them in an ascending fashion, and
** then a descending fashion.  This rise
** and fall of pitch will occur for each
** instrument voice selected by the
** outer loop.
** 1) set looping condition for note rise
** 2) turn note on using value in 'note_val'
** 3) A short delay is called so the
**    notes may be heard ('C' on a 68000
**    will prove fast in a human time frame )
** 4) the note is then turned off
*/
```

```
      for( note_val = 36; note_val < 96; ++note_val ) {
      note_on( 0, note_val, 64 );
      delay();
      note_off( 0, note_val, 64 );
          }

/*
** Note the '--note_val':  why did we use it instead
** of 'note_val--'?  The answer is that there is
** a bug in the version of megamax we are using
** which will not implement a 'variable--'.  The
** bug is now known and I'm sure will be fixed
** by the time you read this.  Megamax is currently
** our favorite 'C' compiler.  The Atari 20 meg.
** hard drive and Megamax are a combination made
** in heaven.
*/

      for( note_val = 96; note_val > 35; --note_val ) {
      note_on( 0, note_val, 64 );
      delay();
      note_off( 0, note_val, 64 );
      }

      }

/*
** The 'appl_exit()' call allows us to
** return to an unblemished desktop
*/

   appl_exit();
}

/*
** end of 'scaler.c'
*/
```

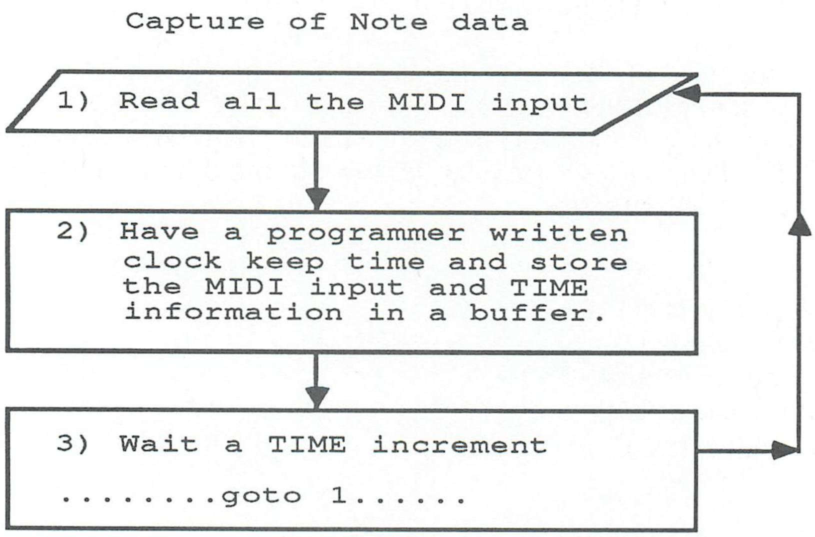
You can change the speed of program execution by changing the values in the delay function. Lower numbers translate into a shorter delay, and that translates into faster play. The music player in `'mdrive.c'` (presented in Chapter IV) is just a simple embellishment of `'scaler.c'`.

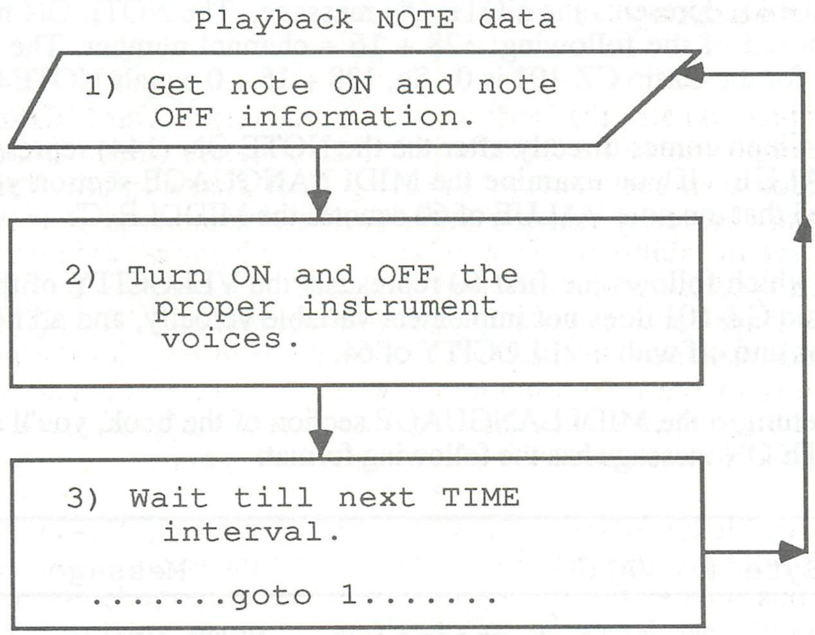
Section 3.4 of Chapter III will demonstrate how you can read the keyboard synthesizer's keyboard, and capture the note on/off information with your Atari ST.

3.4 `readkey.c` source code

This chapter contains the Dorfman/Young challenge. Our MIDI-oriented music composition ST MUSIC BOX lacks a feature which certain MIDI aficionados would be interested in having: Real Time Capture. This MIDI buzz phrase basically means that the software running on your Atari ST would have the ability to record what songs are being played on the MIDI keyboard synthesizer and later play them back. Our reasons for not coding the Real Time Capture option on the initial version of ST MUSIC BOX as explained elsewhere in the book, had nothing to do with the technical complexity of the programming assignment or the equipment. By not programming a Real Time Capture option on our ST MUSIC BOX disk, we have left the avenue open for YOU to write the program.

The source file '`readkey.c`' is a short demo file which shows you how to read the MIDI keyboard and print the information to the screen. You can modify this code so that the information which is currently directed to the screen could be channeled into buffers--for future playing. A VERY SIMPLE algorithm for a Real Time Capture program might look something like this:


Capture of Note data
1) Read all the MIDI input
2) Have a programmer written clock keep time and store the MIDI input and TIME information in a buffer.
3) Wait a TIME increment
........goto 1......


Playback NOTE data
1) Get note ON and note OFF information.
2) Turn ON and OFF the proper instrument voices.
3) Wait till next TIME interval.
.......goto 1.......

If you decided to be really ambitious, then you might look at the 'interf.c' file presented in Chapter IV and convert the Real Time Capture information into a file structure which would be compatible with the AUTO-PLAYER program presented in this volume.

Before we move on to the source code for 'readkey.c' we will discuss the nature of the information which will be presented on the screen. If you were running the program and you pressed the MIDDLE 'C' key on the Casio CZ-101 keyboard synthesizer, you would see the following message printed to your video display:

```
MIDI transmitted 144
MIDI transmitted 60
MIDI transmitted 64
MIDI transmitted 60
MIDI transmitted 0
```

Let's see if we can make some sense of the byte stream received by the Atari ST. We'll take them one value at a time.

The first 144 represents the NOTE ON message. The NOTE ON message is composed of the following: 128 + 16 + channel number. The default channel for the Casio CZ-101 is 0. So, 128 + 16 + 0 equals NOTE ON.

The 60 which comes directly after the the NOTE ON (144) represents the note VALUE. If you examine the MIDI LANGUAGE section you'll be reminded that the note VALUE of 60 denotes the MIDDLE 'C'.

The 64 which follows the first 60 represents the VELOCITY of the note. The Casio CZ-101 does not implement variable velocity, and all notes are turned on and off with a VELOCITY of 64.

If you return to the MIDI LANGUAGE section of the book, you'll see that the NOTE ON message has the following format:

Byte #	VALUE	Message
1)	128 + 16 + channel #	NOTE ON
2)	0 - 127	NOTE PITCH VALUE
3)	0 - 127	NOTE VELOCITY*

*Velocity = 0 -> NOTE OFF

The first three values transmitted from the Casio CZ-101 through the MIDI interface are 144, 60, 64. These values are the NOTE ON message for MIDDLE 'C' in channel 0.

The fourth value 60 actually appeared when you RELEASED the key. The 60 and the 0 actually can be read as a packet of two bytes. The number following the NOTE PITCH VALUE, the VELOCITY, will turn an instrument voice OFF when it is 0. That's correct. We'll repeat it again. If the VELOCITY of a NOTE is '0', whether it is sent with a NOTE ON or a NOTE OFF message, will turn a note OFF.

A new status byte (>= 128...see MIDI LANGUAGE) need not be sent as the status of the keyboard synthesizer had not changed. Let's review values sent from the MIDI one more time:

```
Turn ON MIDDLE 'C'
------------------
MIDI transmitted 144    -> status byte for Note ON
MIDI transmitted 60     -> note value middle 'C'
MIDI transmitted 64     -> turn middle 'C' on velocity 64

Turn OFF MIDDLE 'C'
--------------------
MIDI transmitted 60     -> note value middle 'C'
MIDI transmitted 0      -> velocity '0' means NOTE OFF
```

If we were to explore the algorithm for a Real Time Capture program in a bit more depth, we might create the following rules for MIDI received information:

1) If a received value is 144 -> (144+16) there is a NOTE ON message in a channel (value - 144)

2) The value directly following the NOTE ON message is a note PITCH value ranging from 0 to 127.

3) The value after the PITCH value represents VELOCITY. If the velocity is equal to '0' it will turn the NOTE off.

4) The PITCH value and Velocity value will alternate until a new STATUS byte appears.

To test the information presented, run the program and press the MIDDLE 'C' key, but do not release it. Look at your video display. If you are using a Casio CZ-101 (if you're not using a CZ-101 the VELOCITY number will vary) you will see:

```
MIDI transmitted 144
MIDI transmitted 60
MIDI transmitted 64
```

Release the MIDDLE 'C' key and two more bytes will be printed to the screen. They are:

```
MIDI transmitted 60
MIDI transmitted 0
```

If you do write the Real Time Capture program it might turn into an article, or more. We'd love to see any MIDI-related programs you might write as a result of reading this text. You may send them to us at Abacus.

On to the source code for 'readkey.c'.

```
/*
** readkey.c
**
** This file will demonstrate how
** read keyboard data transmitted
** from the midi synthesizer
** to the Atari ST.  It was compiled
** and linked using Megamax C.
*/

/*
** include files here
*/

#include  <osbind.h>
#include  <stdio.h>

/*
* globals here
*/

int i2[6], mkmx, mkmy, mkmstate, mkkstate;
int i3[6], holdv;
int gflag, gflag1, gflag2, toggle, ctr;


/*
** global variables and arrays for GEM
*/
```

```
int  intin[ 256 ], intout[ 256 ], ptsin[ 256 ];
int  ptsout[ 256 ], contrl[ 12 ];
int  handle, dummy;

/*
** initialize gem
*/

init_gem()
{
int i;
    for( i = 0; i < 10; i++ )
        intin[ i ] = 1;
    intin[ 10 ] = 2;
    handle = graf_handle( &dummy, &dummy, &dummy, &dummy );
    v_opnvwk( intin, &handle, intout );
}

/*
** Short delay
*/

delay()
{
int d1, d2;
    for( d1 = 0; d1 < 1000; d1++ ) {
        for( d2 = 0; d2 < 10; d2++ )
            d2 = d2;
        }
}


/*
** get an int from the midi
** and return it
*/

from_midi()
{
int ifm;

    ifm = Bconin( 3 );
    ifm &= 0x00ff;
    printf("MIDI trasnsmitted byte %d \n", ifm );

    return( ifm );
}


/*
** check to see if a midi byte is waiting
** returns a 0 if no and a 1 if yes
*/

midi_status()
{
int ims;
    ims = Bconstat( 3 );
    if ( ims != 0 )
        ims = 1;
    return( ims );
}


/*
* check the status of midi out and evaluate
*/

check_status()
{
int flag, ret;
    flag = midi_status();
    switch( flag ) {
        case  0:
            gflag = 1;
            break;
        default:
            ret = from_midi();

            break;
        }
    return( ret );
}
```

```
/*
** clean out all the midi bytes
** received from cz 101
*/

clean_house()
{
    gflag = 0;
    while( gflag == 0 ) {
        check_status();
        }
}

/*
* wait on a key press
*/

wait_key()
{
char c;
    printf("\n***Waiting...Press RETURN KEY to
continue\n\n");
    c = getchar();
}

/*
** main program code
*/

main()
{
char   *ch, ch1;
int    i1, hold, temp, *voice;

/*
** GEM initialization
*/

    appl_init();
    init_gem();

    v_hide_c(handle); /* hide cursor */
    v_clrwk(handle);  /* clear screen */
```

```
/*
** program introduction
*/

printf("This program, by Len Dorfman and Dennis Young,\n");
printf("will read data transmitted to the Atari ST\n");
printf("through the MIDI ports.\n");


   wait_key();

printf("Bytes are transmitted from the MIDI when there\n");
printf("are MIDI keyboard events.  If the transmitted \n");
printf("information is not received by the Atari ST it\n");
printf("will remain stored. To demonstrate how the\n");
printf("MIDI transmits note information we will clean\n");
printf("out the MIDI of all bytes.");
   wait_key();

   clean_house();

printf("The MIDI pipe is now clean and you may press\n");
printf("any key on the MIDI keyboard.  To return to the\n");
printf("desktop press the CONTROL key.\n\n");

/*
** We use the GEM 'graf_mkstate' call
** to leave the MIDI read loop because
** the program will cycle through the
** call without stopping.  Remember
** to link the GEM libraries in your
** compiler package in order to have
** the program link properly.
**
** The mkkstate is short for 'mouse
** /keyboard state'.  You can read
** the CONTROL, ALTERNATE, LEFT SHIFT,
** and RIGHT SHIFT keys.  If any of
** them are pressed the 'graf_mkstate'
** call will put a non-zero value in
** variable mkkstate.  The while loop
** will then terminate.
*/
```

```
    mkkstate = 0;
    while( mkkstate == 0 ) {

       graf_mkstate( &mkmx, &mkmy, &mkmstate, &mkkstate );

       i1 = midi_status();  /*is MIDI byte waiting*/
       if( i1 != 0 )        /* if YES then */
          from_midi();       /* print the value to screen */

       }


    v_show_c( handle, 1 );

    appl_exit();
}

/*
** end of readkey.c
*/
```

Section 3.5 will present the source code for a small program which will demonstrate the SYSTEM EXCLUSIVE call on a Casio CZ-101 electronic synthesizer.

Example Screen Display of `READKEY.PRG`:

```
This program written by Len Dorfman and Dennis Young,
will read data transmitted to the ATARI ST through the
MIDI ports.

***Waiting...Press RETURN KEY to continue

Bytes are transmitted from the MIDI when there are
MIDI keyboard events.  If the transmitted information
is not recieved by the ATARI ST it will remain stored.
To demonstrate how the MIDI transmits note information
we will clean out the MIDI of all bytes.

***Waiting...Press RETURN KEY to continue


MIDI transmitted byte 0
The MIDI pipe is now clean and you may press
any key on the MIDI keyboard.  To return to the
desktop press the CONTROL key.

MIDI transmitted byte 144
MIDI transmitted byte 60
MIDI transmitted byte 64
MIDI transmitted byte 60
MIDI transmitted byte 0
MIDI transmitted byte 144
MIDI transmitted byte 61
MIDI transmitted byte 64
MIDI transmitted byte 61
MIDI transmitted byte 0
MIDI transmitted byte 144
MIDI transmitted byte 36
MIDI transmitted byte 64
MIDI transmitted byte 36
MIDI transmitted byte 0
```

3.5 `patch.c` source code

The source code for `'patch.c'`is presented in this section of Chapter III. The program generated by `'patch.c'` will demonstrate how to enable a MIDI SYSTEM EXCLUSIVE call on the Casio CZ-101 synthesizer. The nature of the system exclusive byte sequence is described in your synthesizer's MIDI SYSTEM EXCLUSIVE documentation. If the system exclusive documentation didn't come with your synthesizer, write the manufacturer. They will send you the appropriate information.

Returning to a previously mentioned term, the `PATCH` is a collection of tone data which programs the synthesizer to sound in differing ways. Patches may be programmed into the synthesizer by playing with the appropriate buttons and dials on the synthesizer's body.

The program `'patch.c'` will take a patch which has previously been programmed into the Casio CZ-101 and dump it to your screen or printer. You will be able to explore the patches for the Preset voices, the Internal voices, or even Cartridge voices.

By examining the techniques used in this program, you will be able to extend it so that `'patch.c'` will be able to LOAD and SAVE patches to and from your synthesizer.

One other note: Even though bytes of information are sent and received from the Casio, the system exclusive TONE DATA are split into 1/2 byte statements, called `'nibbles'`. We will briefly describe a 'nibble'.

```
----------------------------------------------------
NIBBLES
----------------------------------------------------

Binary Byte                 high nibble low nibble
---------------             ----------- ----------
1 1 1 1 0 1 0 1             1 1 1 1     0 1 0 1

----------------------------------------------------
```

From the little diagram you can see that bits 7, 6, 5, and 4 are transformed into what's called the 'high order' nibble. Bits 3, 2, 1, and 0 are combined into the 'low order' nibble. Nibble values range from 0 to 15. That is because the 'high order' nibble is evaluated as if the bits were shifted 4 places to the right. The 'high order' nibble would be evaluated like this: 0 0 0 0 1 1 1 1.

In the previous example the 'high order' nibble is 15 and the 'low order' nibble is 5. The Casio outputs 'low order' nibble and then the 'high order' nibble.

The SYSTEM EXCLUSIVE documentation should be much, much more thorough than is presented here. The purpose of this code is to give Casio users a useful program and others a framework to develop programs which use SYSTEM EXCLUSIVE calls on their synthesizers.

```
/*
**   patch.c - by Len Dorfman and Dennis Young
**
**   patch.c, using MIDI exclusive calls
**   will tell the Casio Cz 101 to dump
**   its tone data to the screen
**   for preselect instrument voice 1.
**
**   The following table describes the byte
**   handshake sequence that the Casio systems
**   exclusive call for the SEND REQUEST 1
**   message requires.  (All the values
**   in the following table are HEXADECIMAL).
**
**   Computer -> CZ 101
**   ------------------
**   0xf7, 0x44, 0x00, 0x00, 0x70, 0x10, 0x10
**
**   Cz 101 -> Computer
**   ------------------
**   0xf0, 0x44, 0x00, 0x00, 0x70, 0x30
**
**   Computer -> CZ 101
**   ------------------
**   0x70, 0x31
**
**   CZ 101 -> Computer
**   ------------------
**   **tone data**, 0xf7
```

```
**
**  Computer -> CZ 101
**  ------------------
**  0xf7
**
**  The previous byte sequence will
**  be more full explained in the
**  following source.
*/

/*
**  include files here
*/

#include  <osbind.h>
#include  <stdio.h>

/*
* globals here
*/

int i2[6];
int i3[6], holdv;
int gflag, gflag1, gflag2, toggle, ctr;


/*
**  global variables and arrays for GEM
*/

int intin[ 256 ], intout[ 256 ], ptsin[ 256 ];
int ptsout[ 256 ], contrl[ 12 ];
int handle, dummy;

/*
**  initialize gem
*/

init_gem()
{
int i;
    for( i = 0; i < 10; i++ )
       intin[ i ] = 1;
    intin[ 10 ] = 2;
    handle = graf_handle( &dummy, &dummy, &dummy, &dummy );
    v_opnvwk( intin, &handle, intout );
}

/*
**  Short delay
*/

delay()
{
int d1, d2;
    for( d1 = 0; d1 < 1000; d1++ ) {
       for( d2 = 0; d2 < 10; d2++ )
          d2 = d2;
       }
}

/*
**  Casio patch integer array
*/

int patch[128];

/*
* CASIO system exclusive data
* element:
* 4 is 0x70 + channel #
* 6 is program #
*/


int sendr1[8] = {
    0xf0,0x44,0x00,0x00,0x70,0x10,0x10 } ;

/*
**  send a int to the midi
**
**  the byte will be shown on the
**  screen in hex. format.
*/
```

```
to_midi( itm )
int itm;
{
     Bconout( 3, itm );
     printf("Computer transmitted byte 0x%x \n", itm );
}

/*
**   get an int from the midi
**   and return it in hex. format
**
**   the bit mask 0x00ff will
**   take an integer (which is
**   composed of two bytes) and
**   lop off the upper byte.
**
**   the mask is just a safety
**   procedure--extra cautious
*/

from_midi()
{
int ifm;

     ifm = Bconin( 3 );
     ifm &= 0x00ff;

     printf("MIDI trasnsmitted byte 0x%x \n", ifm );

     return( ifm );
}


/*
**   Get an int from the midi
**   and return it to the
**   calling function.  This
**   call does not print the
**   MIDI byte to the screen.
*/
```

```
p_midi()
{
int ifm;

    ifm = Bconin( 3 );
    ifm &= 0x00ff;

    return( ifm );
}

/*
**  check to see if a midi byte is waiting
**  returns a 0 if no and a 1 if yes
*/

midi_status()
{
int ims;
    ims = Bconstat( 3 );
    if ( ims != 0 )
       ims = 1;
    return( ims );
}


/*
* check the status of midi out and evaluate
*/

check_status()
{
int    flag, ret;
    flag = midi_status();
    switch( flag ) {
       case  0:
           gflag = 1;
           gflag1 = 0xf7;
           break;
       default:
           ret = from_midi();

           break;
       }
    return( ret );
}


/*
* check the status of midi out and evaluate
*/

mp_status()
{
int    flag, ret;
    flag = midi_status();
    switch( flag ) {
       case   0:
          gflag = 1;
          printf("MIDI EMPTY!!!\n");
          break;
       default:
          ret = p_midi();

          break;
       }
    return( ret );
}

/*
**  clean out all the midi bytes
**  received from cz 101
*/

clean_house()
{
    gflag = 0;
    while( gflag == 0 ) {
       check_status();
       }
}

/*
* wait on a key press
*/
```

```
wait_key()
{
char c;
    printf("\n***Waiting...Press RETURN KEY to
continue\n\n");
    c = getchar();
}

/*
**  Converts a digit from decimal to
**  hex. ascii.
*/

conv_itoa( val )
int val;
{
    if( val < 10 )
      val += 48;    /* hex digit to ascii */
    else if ( val == 10 )
      val = 'a';
    else if ( val == 11 )
      val = 'b';
    else if ( val == 12 )
      val = 'c';
    else if ( val == 13 )
      val = 'd';
    else if ( val == 14 )
      val = 'e';
    else
      val = 'f';
    return( val );
}

/*
**  This code will print the patch[] to the screen
**  or printer.
**  1) print message
**  2) set device=0 for printer
**     device = 2 for screen.
**  3) convert patch[] which holds int data
**     into ascii data.
**  4) use bios to print to screen or printer
*/
```

```
char   p_msg[] = "Patch for Instrument voice #" ;

print_patch()
{
int device, column, i2;

    if( toggle == 1 )
       device = 0;   /* printer selected */
    else
       device = 2;   /* screen selected */

    if( device != 0 ) {
       printf("\n\n");
       for( i2 = 0; i2 < 28; ++i2 )
           printf("%c", p_msg[ i2 ] );
           printf("%d", holdv);
       }
    else {
       Bconout( device, '\n' );
       for( i2 = 0; i2 < 28; ++i2 )
           Bconout( device, p_msg[ i2 ] );
           holdv = conv_itoa( holdv );

           Bconout( device, holdv );
             }

    if( device == 0 )
       Bconout( device, '\n' );
    else
       printf("\n");

    for( i2 = 0; i2 < 128; ++i2 )
       patch[ i2 ] = conv_itoa( patch[ i2 ] );
/*
**  column is set for formatting the
**  output to the screen or printer
*/

    column = 0;
    for( i2 = 0; i2 < 128; ++i2 ) {
       Bconout( device, patch[ i2 ] );
       Bconout( device, ',' );
```

```
        ++column;
        if( column > 31 ) {
            column = 0;
            if( toggle == 1 )
                Bconout( device, '\n' );
            else
                printf("\n");
            }
          }
        Bconout( device, '\n' );
        Bconout( device, '\n' );
}

/*
**   main program code
*/

main()
{
char   *ch, ch1;
int i1, hold, temp, *voice;

/*
**   Gem initialization
*/

     appl_init();
     init_gem();

     v_hide_c(handle);/* hide cursor */
     v_clrwk(handle);
     /* clear screen */


/*
**   program introduction
*/

printf("This program, by Len Dorfman and Dennis Young,\n");
printf("will dump the tone data (patch) for a Casio CZ\n");
printf("101 electronic keyboard synthesizer to the\n");
printf("screen or printer.\n\n");
```

```
/*
**   query output
*/

printf("Do you wish to dump to (S)creen or (P)pinter? ");

     ch = &ch1;
     scanf("%c", ch);

/*
**   feedback for choice
*/

     if( ch1 == 'p' || ch1 == 'P' ) {
        toggle = 1;   /* printer */
        printf("\nYou have selected the printer.\n");
        }

     else  {
        toggle = 0;   /* screen if not 'p' */
        printf("\nYou have selected the screen.\n");
        }

     printf("\n\n");

/*
**   the for( ;; ) is an infinite
**   for loop which will be broken
**   out of if a value between 1
**   and 96 is entered
*/

     for( ;; ) {
     printf("Which instrument voice (1 - 96)?");

        voice = &i1;
        scanf("%d", voice );

        if( i1 >= 1 && i1 <= 96 )
           break;
        }
```

```
/*
**  feedback for choice
*/

printf("\nYou have selected instrument voice #%d.\n", i1);

/*
**  save instrument voice
**  selected for later
*/

    holdv = i1;

/*
**  decrement because internal
**  use of instrument voice goes
**  from 0 - 95 and not from
**  1 to 96
*/

    --i1;

    printf("\n");

printf("Make sure that your MIDI cables are connected\n");
printf("to your Casio and that it is turned on.\n\n");
printf("If you are dumping to the printer then make\n");
printf("sure the printer is connected and turned on.\n\n");
printf("The bytes from the MIDI will be remembered in a\n");
printf("buffer so we will clean out the MIDI read\n");
printf("buffer before we to dump the patch.\n");

    wait_key();


/*
**  empty the MIDI buffer
**  of all bytes
*/

    gflag = 0;
    while( gflag == 0 )
```

```
        {
        check_status();
        }


printf("We send this byte to tell the MIDI that we will\n");
printf("begin tranmission of system exclusive ");
printf("information.\n\n");

/*
**   for saftey
*/

     to_midi( 0xf7 );  /* end sys exc */

     wait_key();

     printf("\n");

printf("The SEND REQUEST 1 SYSTEM EXCLUSIVE MESSAGE\n");
printf("is composed of the following hex. bytes:\n\n");
printf("f0 / 44 / 00 / 00 / 70 / 10 / instr. voice #.\n\n");
printf("Let's send them to the Casio.\n");

     wait_key();

/*
**   you change the instrument
**   voice selected in the
**   appropriate spot in the
**   buffer which will soon
**   be sent to the MIDI
*/

     sendr1[ 6 ] = i1;

     printf("Send Request 1\n");
     printf("--------------\n");

/*
**   send the MIDI system exclusive
**   from the buffer to the MIDI
*/
```

```
    for( i1 = 0; i1 < 7; i1++ )
      to_midi(  sendr1[ i1 ] );

    printf("\nThe Casio will now respond with the\n");
    printf("following bytes:\n\n");
    printf("f0 / 44 / 00 / 00 / 70 / 30 .\n\n");
    printf("If the MIDI transmitted bytes match those\n");
    printf("above then the first part of the handshake\n");
    printf("has been successfully completed.  Let's\n");
    printf("see what bytes the MIDI has transmitted.\n");

    wait_key();

/*
**  check to see if the Casio
**  has received the bytes and
**  confirms that it has with
**  the appropriate message
*/

    for( i1 = 0; i1 < 6; i1++ )
      /*  read midi response */
      check_status();

    wait_key();

printf("The ST must transmit the following bytes:\n");
printf("70 / 31.  These hex. bytes signify that the ST\n");
printf("has read the first hand shake response and is\n");
printf("ready to receive the patch tone data.\n");
printf("Let's send the bytes.\n");

    wait_key();

/*
**  more Casio protocol here
*/

    to_midi( 0x70 );
    to_midi( 0x31 );

printf("\nNow that the 70 / 31 hex. bytes have been\n");
printf("transmitted to the MIDI let's get on with the ");
printf("patch  grab!\n");

    wait_key();

/*
**  here is where we grab the
**  patch from the MIDI and place it in
**  the buffer called 'patch[]'
*/
    for(i1 = 0; i1 < 128; i1++ )
      patch[i1] = mp_status();

/*
**  not that we have the patch safely
**  tucked away we can print it
**  to either the screen or printer
**  depending on what we had previously selected
*/

    print_patch();

    wait_key();


/*
**  the 'bye-bye' Casio sign
**  off protocol
*/

    to_midi( 0xf7 );

/*
**  show the cursor and clean up
**  the gem stuff for re-entry
**  to the desktop
*/

    v_show_c( handle, 1 );

    appl_exit();
}
```

```
/*
**   end of patch.c
*/
```

In this chapter we have demonstrated how to send note information from the Atari ST to the MIDI synthesizer and how to read information from the MIDI synthesizer and manipulate with the Atari ST. The programs in this chapter functioned as our introduction to the AUTO-PLAYER.

The program presented in Chapter IV is massive (although it is 1/3 the size of our ST MUSIC BOX). Onward to Chapter IV.

Chapter IV

ST MUSIC BOX AUTO-PLAYER

ST MUSIC BOX AUTO-PLAYER

This chapter will contain the full source code to the ST MUSIC BOX AUTO-PLAYER version 1.0, and various smatterings from the source to XLent Software's ST MUSIC BOX program, which we co-authored. We present this code in an effort to share what we have learned about programming on the ST, working cooperatively as a team and bringing a commercial program from idea to fruition. For some, this personal discussion may not seem germane to learning to program your Atari ST's MIDI port, but for others it may prove instructive reading. We always love to hear how other programming artists and engineers formulate their ideas and work; it expands our creative vision. It is with this hope that we recount our story in words and code.

4.1 In the Beginning

Once we decided to take on the MIDI programming project for XLent Software we decided to delve deeper into MIDI-land and see what existing MIDI software offered. Also we spoke with a few people about what they wished to see in MIDI programs. The program functions were:

1) the ability to enter music note data using the mouse;
2) the ability to play that data through a MIDI STANDARD DEVICE;
3) real time keyboard note capture;
4) the ability to dump the manuscript to a graphic printer.

Xlent Software in concert with our current philosophy decided that writing a product that would come to market for under $50 U.S. would be critical to the program's commercial success. It seemed obvious that a $50 dollar program would not have the same power as a $300 program, but we were intent on providing a powerful package nonetheless. With that in mind we decided to develop a plan which would allow us to approximate how long it would take us to write MIDI related code. Much of the programming initially appeared (and subsequently proved) technically trivial, but we knew from our current list of eight co-authored published programs, "If something can 'gang aft agley', it will!"

We decided that the main program should basically be composed of two elements: a smart note editor and a music player. A separate printer-only dedicated program would be constructed using the editor's file structure and be placed on ST MUSIC BOX's main diskette. The editor and player should handle 8 voices, with two voices assigned to each channel. It seemed clever to have the player simultaneously drive the MIDI keyboard synthesizer and the console speaker at the same time. The focus of the program would be MIDI oriented, but adding the option of the console speaker play would just add a few bytes to what seemed like a clean and simple program (the final code for the ST MUSIC BOX program weighed in at approximately 120,000 bytes).

We decided that the PLAYER should have an animated display. Here Len argued strongly for a clean-as-a-bean diagnostic-type display. No colors or flash. Others in the company argued that colors and flash sell, but Len stood stubborn ground saying, "The player must sound right and visually aid the composer in the sound parameter editing process. There should be nothing that would distract the eye." The view of the company was born out of a COMDEX demonstration where some press commented that they thought the player might be a bit more colorful. When the people at Xlent reported that finding to Len he replied, "I've entered thousands of notes! Not them! I don't want my eyes distracted in any way. Period." And so it goes...

To that end, the PLAYER wound up having animated shapes that danced horizontally as they represented the frequency of the notes being played. The measure being played is displayed by number, as are the INSTRUMENT VOICES assigned to each channel. Dennis reasoned that the listener should be able to stop the composition at any time, examine all the INSTRUMENT VOICES, take notes, and resume playing the piece. All those qualities were eventually built into the ST MUSIC BOX's PLAYER.

The COMDEX computer show deadline loomed on the horizon and we needed a minimum demo for the show. With that in mind we coded the music driver (most of what you will see in the file '`mdrive.c`') first and then we hand assembled (grunt work) in the note data to John P. Sousa's "Stars and Stripes Forever". Keying in all the voices using our assembler was quite tedious, but it was worth the effort because it gave us note data which we eventually used to tune the music driver. The demo was readied in about ten days and we shipped it off to Xlent Software feeling that we had produced something "showable".

Eventually we worked into the player the ability to control the music's speed (tempo) per measure and also we coded in the ability to change the instrument voice per measure. We thought that these were two very nifty features which would add a great deal to the musicality of the player. When we tried out those features on our Casio CZ series synthesizers we were delighted with the results. We could take a single line of music and have a flute play measure 1, a trumpet measure 2 and a whistler for measure three, etc.

You can imagine our SHOCKING surprise when Len demoed the program at a New York user's group and discovered that a Kawai MIDI synthesizer delayed the play at the ONSET of EVERY measure as the PROGRAM CHANGE message was sent from the ST to the Kawai! In New York slang, "What a bummer!" Our nifty little idea of changing the instrument voice per measure using the PROGRAM CHANGE message turned an elegant feature into a headache. So much for the MIDI STANDARD and manufacturer implementations. We began a last minute code fix.

Once we had finished the COMDEX demo we began the planning of the editor. Basically we decided that there would be two approaches we would consider: 1) moving the cursor over the music staffs and a click would place the note on the screen or 2) having the cursor move about a piano-like keyboard and clicking on a note would pop it up on a staff. We bandied about what to do for a while. There were pluses and minuses for both methods.

On the surface, moving the cursor over the staff seemed the more elegant method, and some successful music programs on other computers used that method successfully. Len strongly argued against the more elegant looking method for a variety of reasons. He felt that having the opportunity to place the notes anywhere (horizontally or vertically) on the staffs was too much freedom for him. He felt that a more tightly structured method of note entry would facilitate with less "typos". Dennis brought up the notion of making the editor, in a few clever ways, syntax smart.

We thought that reviewers might take us to task on our plain-Jane method of note entry, but we decided to follow our instincts and not worry about how reviewers might judge the editor's architecture. After all, our series of Printware Programs for Xlent Software on both the XE and ST Atari computers had been reviewed in approximately two dozen sources. These reviews ranged from being printed in large widely circulated magazines to small user group newsletters. It is abundantly clear that when programmers or builders of any engineering art form display their work in a public forum, most often the reviewers reaction will be mixed. What pleases some drives others nuts! Different strokes. We believe that the purest way to develop a product is to temper our creative vision with the opinions of those we respect.

Eventually we chose the piano-like keyboard method of note entry. Here were some of the reasons that come to mind: 1) some people play the piano by ear and don't read music that well; 2) the cursor would have MUCH wider range for note entry - in theory, meaning fewer mechanical errors; and 3) there would be LESS travel between the keyboard for note entry and other musical qualities, such as duration icons.

We have been programming as a team for over four years and know our strengths and weaknesses well. A great deal of our programming and Atari ST knowledge overlaps, so we delegate sections of code according qualities of challenge and fun. Each of us winds up with 'new and challenging code', that which we deem as 'fun code', and the loathesome 'grunt code'. In balance, our efforts are synergistic efforts.

A rough editor was programmed quickly and then the ardous process of entering thousands of notes began. We refined both the player and editor during dozens of test hours. After a few weeks we realized that we had used the editor to create eight songs. When Len keyed in all the notes to a favorite Bach four voice fugue he realized that his and Dennis' vision for the editor worked very well. It was a breeze to enter the contrapuntal music which was enveloped in complex rhythmic patterns. It was a breeze to edit the note data and instrument voices. It was at that moment we had a truly solid product for the MIDI minded Atari ST owner.

At that time we were about 7 days before the COMDEX computer show and decided we could upgrade the previous demo we had sent to Xlent Software and write the AUTO-PLAYER for the COMDEX show. We reasoned that having a program which would play song after song, as a long play record might, would make for a better demonstration than the process of selecting one song after another and loading it into the Atari ST's RAM. The sales person at the COMDEX booth could just turn on the AUTO-PLAYER and concentrate on the process of writing orders and greeting industry pundits.

So was born the ST MUSIC BOX AUTO-PLAYER. We programmed at warp speed getting the program to work. We weren't concerned about making the code as compact as possible: a fully operational demo for COMDEX was our goal.

When the editors at Abacus Software asked us to write a book about the Atari ST and MIDI, we decided to use ALL the code in the AUTO-PLAYER and NOT cull out the fat. Here, again, we decided to not worry about how 'C' wizard reviewers would evaluate the code. Including extra routines would provide extra tutorial for this book's readers.

One specific example would be the function 'ldsnedk()' in the source file 'sets.s'. The function 'ldsendk()' could have been pulled from the code as it does not have anything to do with the AUTO-PLAYER, but it does show the reader one way to read the ST's mouse to determine which non-resource construction set ICON has been selected. The file 'btdata.c' contains specific code presenting two ways to get ICONS up on the screen. As you may have deduced, the ICONS presented are those which appear on the editor screen of Xlent Software's ST MUSIC BOX and the routine 'ldsendk()' determines which non-resource construction set an ICON has been selected. There are other examples of code which we've left from the ST MUSIC BOX intact because we believe it will prove useful to many.

What follows is the 'C' source code for the ST MUSIC BOX AUTO-PLAYER. At the time the AUTO-PLAYER was written we were using the Alcyon compiler supplied in Atari's development package. Currently, we are using Megamax C to create our programs and recommend the Megamax package if your looking for a compiler. The price tag is pretty stiff at $200 retail in the U.S., but combined with some of Abacus' other ST books you'll find yourself, in our opinion, with a faster and more comprehensive C development system than that provided in the Atari package.

On a different front: with the source code in this volume we tried to present a large amount of 'C' source code along with the MIDI information presented earlier. The actual source for the player runs along the lines to approximately 6000 lines of 'C' source, a massive amount to type in. The optional disk which may be purchased along with this book is a very good value if you are thinking of typing in the program. On the disk you'll find: 1) the source typed in!; 2) compiled object files; and 3) MUSIC FILES for a Joplin rag, a Bach fugue, a ditty by F. Couperin ... and more!

So if you plan to take the information presented in this book and start your own MIDI project you probably will not need the disk. If you want to hear the player, the modest price for the disk is an easy way to avoid some intense typing to wind up with no music files!!!

On to the CODE!

4.2 `eio.c` source code

This program was created to demonstrate the capabilities of ST MUSIC BOX and to serve as a mechanism to automatically play pieces created with ST MUSIC BOX. For the most part it is culled from code that already existed in the main program. When we were creating ST MUSIC BOX we used three different C compilers and two assemblers, but most of it was produced with the Alcyon compiler that came with the developer's package. In order to make the code at least reasonably clear we have re-edited the whole hodgepodge and recompiled it with Alcyon. It works, but it certainly isn't pretty. To tell the truth, our code is rarely pretty or even well tuned up. Our attitude in writing programs is that we get the model up and running and return to code only if the 'results' don't satisfy us. We've gotten used to ugly code and we've also gotten used to finishing things on time. Respect your language, but do not pay it homage. How's that for pompous ?

```
/*
**  eio.c
*/

/*
**  include definition files
*/

#include    <a:portab.h>
#include    <a:machine.h>
#include    <a:obdefs.h>
#include    <a:define.h>
#include    <a:gemdefs.h>
#include    <a:osbind.h>

/*
**  declare global variables
*/

int wbox, hbox, dur1, dur2, dur3, port_state;
int dur4,dur5,dur6,dur7,dur8;
int contrl[12], intin[256], ptsin[256], ptsout[256];
int intin[256], intout[256],pxy[10];
int newbyte[8], oldbyte[8], typedone, hbox1, wbox1, hchar1, wchar1;
int l_intin[20], l_out[128], l_ptsin[20];
int ev_kreturn,iter,gr_mkmx,gr_mkmy,groption,stylem,endm,mode,set_mode;
int set_font,handle,dummy,gr_mkmstate,gr_mkkstate,prtmem,set_effect;
int holdx,holdy,whesc,whesc1,color_index,colorm,holdclr;
int gr_mkresvd,intstyle,topbot,height,width,widd,mull,radius;
int begang,endang,yradius,x_boundary,y_boundary,k,k1,k2,k3,k4;
int i,n,u,ab,kk,row,count,column,sp,mod,byte,j,quot,rem,demo;
int note1, note2, note3,note4,note5,note6,note7,note8;
int scrold, scrnew, mult, ymult, findex, fcolor, linev, lineh, linesw;
int byte1, byte2, ig, sgflag,prtsw,path;
int timex, timey, first, octup, octsw0, octsw1, octsw2, meascnt, cnt32;
int octsw3,octsw4,octsw5,octsw6,octsw7;
int chancnt, progcnt, gfreq1, gfreq2, gfreq3;
int gfreq4,gfreq5,gfreq6,gfreq7,gfreq8;
int *ptr1, *ptr2, *ptr3, *ptr4, *ptr5, *ptr6, gpsw, gvibsw, gptime;
int dblo1, *ptr7,*ptr8;
int gprog, progtab[ 512];
int file_handle;
int kreturn;
int exflag;
int holdrez;
int st_m = 1;

int chan1_buffer[264];
int chan2_buffer[264];
int chan3_buffer[264];
int chan4_buffer[264];

int top1_measure;
int top2_measure;
int top3_measure;
int top4_measure;
int top5_measure;
int top6_measure;
int top7_measure;
int top8_measure;

int octave = 4;
int measure = 0;
int key = 1;
int time = 32;
int time_counter = 3;
int key_counter = 1;
int key_sw = 1;

int cm1_pick = 1;
int cm2_pick = 2;
int cm3_pick = 3;
int cm4_pick = 4;

int ch1_meas = 0;
int ch2_meas = 0;
int ch3_meas = 0;
int ch4_meas = 0;

int act1_measure = 1; /* preset 1*/
int act2_measure = 2; /* preset 2*/
int act3_measure = 3; /* preset 3*/
int act4_measure = 4; /* preset 4*/

int this1_measure;
int this2_measure;
int this3_measure;
int this4_measure;

int port1;
int port2;
int port3;
int port4;

int vib1 = 0;
int vib2 = 0;
int vib3 = 0;
int vib4 = 0;

int oct1 = 2;
int oct2 = 2;
int oct3 = 2;
int oct4 = 2;
int oct5 = 2;
int oct6 = 2;
int oct7 = 2;
int oct8 = 2;

int vc1 = 1;
int vc2 = 1;
int vc3 = 1;
int vc4 = 1;
int vc5 = 1;
int vc6 = 1;
int vc7 = 1;
int vc8 = 1;

int key_pick = 25;
int xkey;
int time_check;
int key_check;
int topxbuff[124];
int xkey_input;
int voice;
int io_measure;
int dbuffer[3250]; /* 250 * 13 */
int f;
int f_check;
int tm;
int dcur_drv;
int attr;
int r_voice = 1;
int t,p,yy,y;


long    screen1,screen2,orgscrn,logicbase;
long    ltemp,buff_size,bufferx,l_ptr1;

char    xstr[2] = { '\0','\0' };
char    iobuff1[6336];
char    iobuff2[6336];
char    yspec[5];
char    dta_name[64] = "";
char    xfile_name[64] = "";
char    c;
char    *ptra;
char    dig_buff[128];

    /*------Alcyon does not limit the number of bytes that can
    be designated as variables or allotted in arrays.  With at least
    one of the C compilers available for the ST you are limited to
    32,000 bytes for these purposes.  If you were using that compiler
    the bytes can be reserved with a malloc call (memory allocation)
    which is nicely explained in their documentation.---------------*/


char    buffer1[32767];     /* main display screen */

char    nbuff1[6336];
char    nbuff2[6336];
char    nbuff3[6336];
char    nbuff4[6336];
char    nbuff5[6336];
char    nbuff6[6336];
char    nbuff7[6336];
char    nbuff8[6336];
char    nbuff9[6336];
char    nbuff10[6336];
char    nbuff11[6336];
char    nbuff12[6336];
char    nbuff13[6336];
char    nbuff14[6336];
char    nbuff15[6336];
char    nbuff16[6336];

char    tmpo_buffer[264];


main()
{

/*--------------------Standard  initializations------------------- */

    appl_init();      /*  only one on board - no ID really needed  */


/*--appl_init() will return a value to you to be used as an
    identification number if you're planning on having more
    than one program resident.

    In that case, you would write something like:
        appl_id = appl_init();  -------------------------------*/


    for (i=0; i<10; ++i)
        {
        intin[i] = 1;
        }
    intin[10]=2;
    handle = graf_handle(&width,&height,&wbox,&hbox);

 /*------The graf_handle function writes the letter width,letter height,
    character box width and character box height into the addresses
    you pass it.  It returns a handle that is used each time
    you access the VDI.---------------------------------------*/

    v_opnvwk(intin, &handle, intout);
    v_clrwk(handle);

 /*------Enable clipping function and establish initial parameters---*/

    pxy[0]=0;
    pxy[1]=0;
    pxy[2]=intout[0];
    pxy[3]=intout[1];
    x_boundary = pxy[2]/2;
    y_boundary = pxy[3]/2;
    vs_clip(handle,1,pxy);

/*------We have separate programs for the medium and high resolution
    monitors(none for the low resolution), so we check the Gem
    function Getrez() before proceeding.  This is the color version.

    The returned value are:
    Low resolution = 0;
    Medium resolution = 1;
    High resolution = 2;

    If the correct program / monitor combination is not established
    the user is returned to the desktop.

    Writing      if (Getrez() != 1)      here does the trick nicely
    but we made other reference to screen resolution so we stuck it
    in a variable to avoid another call. --------------------------*/

    holdrez = Getrez();
    if (holdrez != 1)
        {
        v_gtext(handle,width*0,height*3,
        "Xlent's MUSIC BOX Player is for Med Res only!");
        v_gtext(handle,width*0,height*6,
        "PRESS ANY KEY for DESK TOP");
        ev_kreturn = evnt_keybd();

        goto todesk;  /*    Exit to desktop   */

        }

/*------This next bit of code establishes the screen address;
    not all compilers will let you treat the long int
    screen1 in this fashion but Alcyon does.  There are other
    ways of setting the screen address but this was left over
    from another program that had buffers moving in, out, up,
    down, left and right, and we just stuck with it. For your
    information, a simple:
        orgscrn = (long) Physbase();
    or
        orgscrn = (char *)Physbase(); (with some compilers)

    by itself, should get you going nicely.  --------------------*/
    screen1 = (0xfff00&buffer1);      /*top screen addr*/
    screen1 += (0x100);               /*adjust ptr*/
    vsf_perimeter(handle,1);          /*edge of shape visible*/


    v_hide_c (handle);             /* hide mouse pointer */
    v_clrwk(handle);               /* clear the screen */
    timex = 180;                   /* default tempo values */
    timey = 50;                    /*  ---------------      */


    denw2();              /*  a small window for title */

    v_gtext(handle,width*22,height*8,
    "    XLent  Software   presents");
    v_gtext(handle,width*22,height*10,
    " ST MUSIC BOX AUTOPLAYER (c) 1986");
    v_gtext(handle,width*22,height*14,
    "  by  Dennis Young & Len Dorfman");
    v_gtext(handle,width*22,height*15,
    "   Press  any  Key  to continue");
    v_gtext(handle,width*22,height*11,
    "              V.   1.0");
    v_gtext(handle,width*22,height*12,
    "From the Abacus book ATARI ST");
    v_gtext(handle,width*22,height*13,
    "Introduction to MIDI Programming");


    ev_kreturn = evnt_keybd();
    v_clrwk(handle);

    while ( exflag == 0 )
        iowork();

/*-----If resolution wrong or exflag != 0, program exits here. */

todesk:

     vst_height(handle,13,&wchar1,&hchar1,&hbox,&wbox);
     set_effect = vst_effects(handle,0);
     vst_rotation(handle,0);
     vswr_mode(handle,0);
     v_clrwk(handle);          /*clear the screen*/
     v_clsvwk(handle);
}
/*------end main---------------*/
```

```
/*-------This is set up to repeat the whole process 20 times -------*/


iowork()
{
int uuu,jjj,fff,yyy,ccc;

    get1_path();
    for (ccc = 0;ccc < 20; ++ccc)
    {
    for (jjj = 0, uuu = 0, yyy = 0; jjj < tm; ++jjj, ++uuu)

        {
        dcur_drv = Dgetdrv();
        xfile_name[0] = dcur_drv + 'A';
        xfile_name[1] = ':';

        for (fff = 2; fff < 15; ++fff, ++yyy)
            xfile_name[fff] = dbuffer[yyy];


        do_c_load();
        playit();
        }
    }
}

/*
**
*/

ex_mode()
{       yspec[0] = '*';
        yspec[1] = '.';
        yspec[2] = 'M';
        yspec[3] = 'U';
        yspec[4] = 'S';
}



clrdta()
{
    for (f = 30; f < 43; ++f)
            dta_name[f] = 32;
}
```

```
clr_dbuffer()
{
    for (f = 0; f < 3250;  ++f)
            dbuffer[f] = 32;
}

/*
**
*/

char
*qqstr (pd, ps)
char    *pd, *ps;
{
    while (*pd)
        pd++;
    while (*pd++ = *ps++);
        return(pd);
}


/*------This routine bypasses the fsel_input routine that fetches the
    standard Gem IO Window.  It reads into a buffer the names
    of all the files that have the extender .MUS and it can handle
    250 files.  There is no major advantage to doing it this way over
    looping through the directory for each title but with so much
    memory it just felt right.  We've detailed most of the IO code
    since not everyone is very familiar with ST disk work and
    it's easy to get lost in.-----------------------------------*/


getl_path()
{

int     i,y;

    attr = 0x17;
    tm = 0;

    clr_dbuffer();

    ex_mode();
    vst_effects(handle, 0);

    for (i=0; i < 64; i++)
        xfile_name[i] = 0;
```

```
/*------l_ptr1 is a long pointer which is used to pass the address of a
    buffer to which a subsequent call - Fsfirst() - is going to
    write data.  This buffer should be allotted 64 bytes to hold the
    data that's going to be coming back.  It's referred to as the
    disk transfer address----------------------------------------*/

    l_ptr1 = dta_name;
    Fsetdta(l_ptr1);


/*------Dgetdrv returns the number of the current disk drive (0 = A:,
    1 = B:, 2 = C:, etc.).  In order to make the drive designator
    consistent with what's required for other IO work - ASCII is passed
    back and forth - we add the ASCII value for 'A' to the number
    returned. ---------------------------------------------------*/

    dcur_drv = Dgetdrv();
    xfile_name[0] = dcur_drv + 'A';
    xfile_name[1] = ':';

/*------Dgetpath returns the current path or folder into our filename
    buffer.  After that we concatenate the extender.--------------*/

    Dgetpath( &xfile_name[2], dcur_drv +1);
    qqstr(xfile_name,yspec);


/*------Attr is a set of attributes that are to be matched.

                         $01     read only
                         $02     hidden
                         $04     system (hidden)
                         $08     volume label
                         $10     subdirectory
                         $20     written to and closed

    Fsfirst dumps data into the our xfile_name buffer.  The function
    actually calls for the first parameter passed to be a char
    pointer to the buffer. We are only looking for the filename and
    its extender but there is other information provided.


                file attributes      byte 21
                file time stamp      byte 22-23
                file date stamp      byte 24-25
                file size (as a long)    byte 26-29
                name/extender of file    byte 30-43
    A 0 is returned if the file is found.  The loop following Fsfirst()
    starts loading the filenames into a buffer which will eventually
    serve to hold all of the eligible files (those with MUS
    extender).-*/



    ptr1 = xfile_name;
    f_check = (Fsfirst( ptr1, attr ));
        for (i = 0,f = 30; f < 43; ++i, ++f)
            dbuffer[i] = dta_name[f];
    clrdta();



/*------Fsnext() continues the process started by Fsfirst(); it's not
    necessary to pass any parameters - you must only extract whatever
    information you want.  The loop will get as many as 250 names
    and plop them all into dbuffer[3250] :: 250 * 13 bytes each. --*/


    i = 13;    /*  one file already in place from Fsfirst()   */
    do
    {
        f_check = Fsnext();
        if ((f_check == 0) && (tm < 250))
            {
            ++tm;
            for (f = 30; f < 43; ++f,++i)
                dbuffer[i] = dta_name[f];
            clrdta();
            }
    }
    while (f_check==0);
}


clr_iobuff()
{
    for (f = 0; f < 6336;  ++f)
        {
            iobuff1[f] = 0;
            iobuff2[f] = 0;
        }
}
```

```
/*
   We set up these buffers apart from each other to allow
   easier, more explicit access to them.  At various times in the
   program's developement, we would have different buffers and
   counters on the screen to make for easy debugging.  Pointing
   into a single large array is cumbersome in cases where so
   many unlike variables are being tracked.

   The Midi likes to pass 16 bit values and that's the reason we
   used them instead of chars.  The exception is the tmpo_buffer
   which was cut in last - we hadn't left enough room in the header
   to accomodate it as 16 bit numbers so we put it in as chars and
   convert them before the value is passed.
*/


get_counters()
{
        bufferx = &chan1_buffer[0];
        buff_size = 528;
        Fread(file_handle,buff_size,bufferx);

        bufferx = &chan2_buffer[0];
        buff_size = 528;
        Fread(file_handle,buff_size,bufferx);

        bufferx = &chan3_buffer[0];
        buff_size = 528;
        Fread(file_handle,buff_size,bufferx);

        bufferx = &chan4_buffer[0];
        buff_size = 528;
        Fread(file_handle,buff_size,bufferx);

        bufferx = &topxbuff[0];
        buff_size = 248;
        Fread(file_handle,buff_size,bufferx);

        bufferx = &tmpo_buffer[0];
        buff_size = 264;
        Fread(file_handle,buff_size,bufferx);

        top1_measure = topxbuff[0];
        top2_measure = topxbuff[1];
        top3_measure = topxbuff[2];
        top4_measure = topxbuff[3];
        top5_measure = topxbuff[4];
```

```
top6_measure = topxbuff[5];
top7_measure = topxbuff[6];
top8_measure = topxbuff[7];

key = topxbuff[8];
time = topxbuff[9];

cm1_pick = topxbuff[10];
cm2_pick = topxbuff[11];
cm3_pick = topxbuff[12];
cm4_pick = topxbuff[13];
ch1_meas = topxbuff[14];
ch2_meas = topxbuff[15];
ch3_meas = topxbuff[16];
ch4_meas = topxbuff[17];

act1_measure = topxbuff[18];
act2_measure = topxbuff[19];
act3_measure = topxbuff[20];
act4_measure = topxbuff[21];

this1_measure = topxbuff[22];
this2_measure = topxbuff[23];
this3_measure = topxbuff[24];
this4_measure = topxbuff[25];

timex = topxbuff[26];
timey = topxbuff[27];

port1 = topxbuff[28];
port2 = topxbuff[29];
port3 = topxbuff[30];
port4 = topxbuff[31];

vib1 = topxbuff[32];
vib2 = topxbuff[33];
vib3 = topxbuff[34];
vib4 = topxbuff[35];

oct1 = topxbuff[36];
oct2 = topxbuff[37];
oct3 = topxbuff[38];
oct4 = topxbuff[39];
oct5 = topxbuff[40];
oct6 = topxbuff[41];
oct7 = topxbuff[42];
oct8 = topxbuff[43];
```

```
        vc1 = topxbuff[44];
        vc2 = topxbuff[45];
        vc3 = topxbuff[46];
        vc4 = topxbuff[47];
        vc5 = topxbuff[48];
        vc6 = topxbuff[49];
        vc7 = topxbuff[50];
        vc8 = topxbuff[51];

        time_counter = topxbuff[52];
        key_counter = topxbuff[53];
        xkey = topxbuff[54];

}

/*
**
*/

do_c_load()
{
int     check;
int ii;


/*------Fopen wants the file name and a mode value which will be 0
    for read, 1 for write or 2 for read / write; the operation
    returns a number that serves has a file designator. It will
    return a negative number in the event of an error.---------*/

        file_handle = Fopen (ADDR(xfile_name), 2);
        if (file_handle >= 0)
        {
            get_counters();

        for (r_voice = 1; r_voice < 9; ++r_voice)
            {
            clr_iobuff();
            nmove();
            mv_io_n();
            }
```

```
/*------Fclose needs only the file designator to wrap it up -------*/

            Fclose(file_handle);
        }
}

/*
**
*/

nmove()
{
int cc,mm;

/*-------Fread needs three things:

        1.  File designator or handle
        2.  Number of bytes to read stored in a long

        3.  Char pointer to receiving buffer(though Alcyon also
            happily accepts a long with the buffer address as it is
            here)

    The number of bytes actually read is returned so if you need or
    want to know the number, set up the call like this:

    (int or long) number =  Fread(handle,count,buffer);-----------*/



    for(mm = 0; mm < 264 ; ++mm)
    {
        for(cc = 0; cc< 24 ; ++cc)
        {
        bufferx = &xstr[0];
        buff_size = 1;
        Fread(file_handle,buff_size,bufferx);
        if (xstr[0] != 249)
            iobuff1[(mm * 24) + cc] = xstr[0];
        else
            cc = 24;
        }
    }

    for(mm = 0; mm < 264 ; ++mm)
    {
        for(cc = 0; cc< 24 ; ++cc)
        {
        bufferx = &xstr[0];
        buff_size = 1;
        Fread(file_handle,buff_size,bufferx);
        if (xstr[0] != 249)
            iobuff2[(mm * 24) + cc] = xstr[0];
        else
            cc = 24;
        }
    }

}

/*
**
*/

mv_io_n()
{

    switch(r_voice)
    {
    case 0x01:
    for ( yy = 0; yy < 6336; ++yy)
        {
        nbuff1[yy] = iobuff1[yy];
        nbuff2[yy] = iobuff2[yy];
        }
    break;
    case 0x02:
    for ( yy = 0; yy < 6336; ++yy)
        {
        nbuff3[yy] = iobuff1[yy];
        nbuff4[yy] = iobuff2[yy];
        }
    break;
    case 0x03:
    for ( yy = 0; yy < 6336; ++yy)
        {
        nbuff5[yy] = iobuff1[yy];
        nbuff6[yy] = iobuff2[yy];
        }
    break;
    case 0x04:
    for ( yy = 0; yy < 6336; ++yy)
        {
        nbuff7[yy] = iobuff1[yy];
```

```
            nbuff8[yy] = iobuff2[yy];
            }
        break;
        case 0x05:
        for ( yy = 0; yy < 6336; ++yy)
            {
            nbuff9[yy] = iobuff1[yy];
            nbuff10[yy] = iobuff2[yy];
            }
        break;
        case 0x06:
        for ( yy = 0; yy < 6336; ++yy)
            {
            nbuff11[yy] = iobuff1[yy];
            nbuff12[yy] = iobuff2[yy];
            }
        break;
        case 0x07:
        for ( yy = 0; yy < 6336; ++yy)
            {
            nbuff13[yy] = iobuff1[yy];
            nbuff14[yy] = iobuff2[yy];
            }
        break;
        case 0x08:
        for ( yy = 0; yy < 6336; ++yy)
            {
            nbuff15[yy] = iobuff1[yy];
            nbuff16[yy] = iobuff2[yy];
            }
        break;

        default:
        break;
        }
}


denw2()
{
    int pxy8[10];
    setldparam();

    pxy8[0] = 160 - 8;
    pxy8[1] = (128 - 48)/2;
    pxy8[2] = 480;
    pxy8[3] = (128 - 48)/2;
    pxy8[4] = 480;
    pxy8[5] = 272/2;
    pxy8[6] = 160 - 8;
    pxy8[7] = 272/2;
    pxy8[8] = 160 - 8;
    pxy8[9] = (128 - 48)/2;
    pxy[0] = 160 - 8;
    pxy[1] = (128 - 48)/2;
    pxy[2] = 480;
    pxy[3] = 272/2;
    vr_recfl(handle,pxy);
    v_pline(handle,5,pxy8);
    vsf_interior(handle,0);
    pxy8[0] = 168 - 8;
    pxy8[1] = (144 - 48)/2;
    pxy8[2] = 472;
    pxy8[3] = (144 - 48)/2;
    pxy8[4] = 472;
    pxy8[5] = 256/2;
    pxy8[6] = 168 - 8;
    pxy8[7] = 256/2;
    pxy8[8] = 168 - 8;
    pxy8[9] = (144 - 48)/2;
    pxy[0] = 168 - 8;
    pxy[1] = (144 - 48)/2;
    pxy[2] = 472;
    pxy[3] = 256/2;
    vr_recfl(handle,pxy);
    vsl_width(handle,1);
    v_pline(handle,5,pxy8);
}


setldparam()
{
    vsf_color(handle,1);
    vswr_mode(handle,1);
    vsf_interior(handle,2);
    vsl_width(handle,1);
    vsl_color(handle,1);
        vsf_style(handle,1);
}

/*
**  end eio.c
*/
```

4.3 mdrive.c source code

This file is the main music driver file. It will play simultaneously from the console speaker and through the MIDI port.

```
/*  mdrive.c                    */

/*
**  include definition files
*/

#include    <a:portab.h>
#include    <a:machine.h>
#include    <a:obdefs.h>
#include    <a:define.h>
#include    <a:gemdefs.h>
#include    <a:osbind.h>

/*
**  external definitions
*/

extern  int wbox, hbox, dur1, dur2, dur3, port_state;
extern  int dur4,dur5,dur6,dur7,dur8;
extern  int contrl[12], intin[256], ptsin[256], ptsout[256];
extern  int intin[256], intout[256],pxy[10];
extern  int newbyte[8], oldbyte[8], typedone, hbox1, wbox1, hchar1,
wchar1;
extern  int l_intin[20], l_out[128], l_ptsin[20];
extern  int ev_kreturn,i,iter,gr_mkmx,gr_mkmy,groption,stylem,endm;
extern  int mode,set_mode;
extern  int set_font,handle,dummy,gr_mkmstate,gr_mkkstate,prtmem;
extern  int holdrez,set_effect;
extern  int holdx,holdy,whesc,whesc1,color_index,colorm,holdclr;
extern  int gr_mkresvd,intstyle,topbot,height,width,exflag;
extern  int widd,mull,radius;
extern  int begang,endang,yradius,x_boundary,y_boundary,k,k1,k2,k3,k4;
extern  int n,u,ab,kk,row,count,column,sp,mod,byte,j,quot,rem,demo;
extern  int note1, note2, note3, note4, note5, note6, note7, note8;
extern  int scrold, scrnew, mult, ymult, findex, fcolor, linev;
extern  int lineh, linesw;
extern  int byte1, byte2, ig, sgflag,prtsw,path;
extern  int timex, timey, first, octup;
extern  int octsw0, octsw1, octsw2, octsw3, octsw4, octsw5, octsw6;
extern  int octsw7, meascnt, cnt32;
extern  int chancnt, progcnt;
extern  int gfreq1, gfreq2, gfreq3, gfreq4,gfreq5,gfreq6,gfreq7,gfreq8;
extern  int gpsw, gptime, dblo1, gvibsw, gprog;
extern  long    *ptr1, *ptr2, *ptr3, *ptr4, *ptr5, *ptr6, *ptr7, *ptr8;
extern  char    c;
extern  long    logicbase,screen1,screen2,orgscrn,ltemp;
extern  char    buffer1[32767]; /* main display screen */
extern  int fflag1,fflag2,fflag3,fflag4,fflag5,fflag6,fflag7,fflag8;
extern  int oct1, oct2, oct3, oct4, oct5, oct6, oct7, oct8;
extern  int prog0c[264], prog1c[264], prog2c[264], prog3c[264];
extern  int vc1, vc2, vc3, vc4, vc5, vc6, vc7, vc8, time;
int     ldtmpo[ 264 ];

/*------program start-----------------------*/

int gpflag;
int gprog1, gprog2, gprog3;

/*
**  for future use
*/

beep1()
    {
    }

/*-----demo music buffers-----*/


int dtab1[1024], dtab2[1024],dtab3[1024],
    dtab4[1024], dtab5[1024],dtab6[1024],
    dtab7[1024], dtab8[1024];

int ftab1[1024], ftab2[1024],ftab3[1024],
    ftab4[1024], ftab5[1024],ftab6[1024],
    ftab7[1024], ftab8[1024];

int vtab1[1], vtab2[1],vtab3[1],
    vtab4[1], vtab5[1],vtab6[1],
    vtab7[1], vtab8[1];

int progtab;
```

```
/*
**   these values are taken directly from the General
**   Instrument's manual for the sound chip inside the
**   ST.  You need to pass the chip a fine tune value
**   and a coarse tune value in order to produce a
**   legal note frequency.  The ftune and ctune tables
**   will simplify playing music from the consol speaker.
**   The process is to index these arrays and then pass
**   the GI sound chip an index value ranging from 1
**   to 96.
*/

/*--------GI sound chip frequency values----------------*/
/* 50 */
    int ftune[97] = {
        0xff,
        0x5d,0x9c,0xe7,0x3c,0x9b,0x02,0x73,0xeb,0x6b,0xf2,
        0x80,0x14,0xae,0x4e,0xf4,0x9e,0x4d,0x01,0xb9,0x75,
        0x35,0xf9,0xc0,0x8a,0x57,0x27,0xfa,0xcf,0xa7,0x81,
        0x5d,0x36,0x1b,0xfc,0xe0,0xc5,0xac,0x94,0x7d,0x68,
        0x53,0x40,0x2e,0x1d,0x0d,0xfe,0xf0,0xe2,0xd8,0xca,
        0xbe,0xb4,0xaa,0xa0,0x97,0x8f,0x87,0x7f,0x78,0x71,
        0x6b,0x65,0x5f,0x5a,0x55,0x50,0x4c,0x47,0x43,0x40,
        0x3c,0x39,0x35,0x32,0x30,0x2d,0x2a,0x28,0x26,0x24,
        0x22,0x20,0x1e,0x1c,0x1b,0x19,0x18,0x16,0x15,0x14,
        0x13,0x12,0x11,0x10,0x0f,0x0e } ;

    int ctune[97] = {
        0xff,
        0x0d,0x0c,0x0b,0x0b,0x0a,0x0a,0x09,0x08,0x08,0x07,
        0x07,0x07,0x06,0x06,0x05,0x05,0x05,0x05,0x04,0x04,
        0x04,0x03,0x03,0x03,0x03,0x03,0x02,0x02,0x02,0x02,
        0x02,0x02,0x02,0x01,0x01,0x01,0x01,0x01,0x01,0x01,
        0x01,0x01,0x01,0x01,0x01,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00 } ;


/*------set the asdr envelope-------------*/
/*------ft1 -> 0 - 32000------------------*/
/*------ct1 -> 0 - 32000------------------*/
/*------ens -> 0,4,8,9,10,11,12,13,14,15--*/

setenv( ft1, ct1, ens)
```

```
    int ft1, ct1, ens;
    {
    Giaccess ( ft1, 0x8b );      /* fine tune env speed */
    Giaccess ( ct1, 0x8c );      /* coarse tune env speed */
    Giaccess ( ens, 0x8d  );     /* select env shape
    */
    }

/*
**  This function writes to the MIDI
**  and will tell the keyboard synthesizer
**  to change the portamento value. See the
**  MIDI LANGUAGE section on control codes for more
**  infiormation.
*/

/*-----change portimento-----*/

chport ( psw, ptime, pchan )
    int psw, ptime, pchan;
    {
    if ( pchan == 0 )
        gptime = ptime;
    switch ( ptime )
        {
        case 0x00:
            Bconout ( 3, ( 128 + 32 + pchan + 16 ) );
            Bconout ( 3, 65 );  /* 100 */
            Bconout ( 3, 0 );
            break;
        default:
            Bconout ( 3, ( 128 + 32 + pchan + 16 ) );
            Bconout ( 3, 65 );
            Bconout ( 3, 127 );
            Bconout ( 3, ( 128 + 32 + pchan + 16 ) );
            Bconout ( 3, 5 );
            Bconout ( 3, ptime );
            break;
        }
    }

/*-----change portamento-----*/

chvib ( vtime, vchan )
    int vtime, vchan;
    {
    switch ( vtime )
```

```
        {
        case 0x00:
            Bconout ( 3, ( 128 + 32 + vchan + 16 ) );
            Bconout ( 3, 1 );    /* 100 */
            Bconout ( 3, 0 );
            break;
        default:
            Bconout ( 3, ( 128 + 32 + vchan + 16 ) );
            Bconout ( 3, 1 );
            Bconout ( 3, 127 );
            break;
        }
    }

/*
**  This simple function allows you to
**  change the program, or instrument
**  voice for a specific CHANNEL.  On entry
**  it receives the CHANNEL # and then
**  the PROGRAM #.  The instrument corresponding
**  to the program number can be found in your
**  synthesizer's manual.
*/

/*-----set progeram change-----*/

chprog ( chan3, prog )
    int chan3, prog;
    {
    Bconout ( 3, (128 + 64 + chan3 ) );
    Bconout ( 3, prog );
    }

/*
**  This function turns a note off.
**  The reason why the freq-- appears
**  is that the index for the frequency
**  in the player and editor begins with
**  the lowest 'C' having the value of 1.
**  The lowest 'C' on the MIDI synthesizer
**  will be 0, hence the freq--.
*/

/*-----midi note off-----*/

noteoff ( chan1,  vel1, freq1 )
    int chan1, vel1, freq1;
    {

    if ( freq1 <= 36 )
        freq1 = 1;
    else
        freq1--;
    Bconout ( 3, 128 + chan1 ); /* octave */
    Bconout ( 3, freq1 );    /* casio default..temp */
    Bconout ( 3, 0 );        /* freq playing */
    }

/*
**  This function turns a note on.  The
**  if...else if...else code adjusts out
**  of bounds notes to play on the Casio
**  CZ 101.  In this version of the
**  AUTO-PLAYER the default frequency index
**  value for a rest is 1.  The call is
**  passed: first the channel in which to
**  turn he voice on, second, the velocity
**  (or note loudness), and third the index
**  representing the note to be played.
**  Note that the VELOCITY is NOT used in this
**  AUTO-PLAYER as we have chosen the default
**  value of 64.  On future MIDI products we
**  will change the default value to reflect
**  a real volume.  As the ST MUSIC BOX 1.1 does
**  allow the user to change VELOCITY we saw no
**  no reason to use the passed parameter.
*/

/*-----midi note on-----*/

noteon ( chan2,  vel2, freq2 )
    int chan2, vel2, freq2;
    {
    if ( freq2 == 1 )
        freq2 = 1;
    else if ( freq2 <= 12 )
        freq2 += 36;
    else if ( freq2 <= 24 )
        freq2 += 24;
    else if ( freq2 <= 36 )
        freq2 += 12;
    else if ( freq2 >= 132 )
        freq2 -= 36;
    else if ( freq2 >= 120 )
```

```
        freq2 -= 24;
    else if ( freq2 >= 108 )
        freq2 -= 12;
    else
        freq2 = freq2;

    if ( freq2 <= 36 )
        freq2 = 1;
    else
        freq2--;

    switch ( freq2 )
        {
        case 0x01:
            Bconout ( 3, 128 + chan2 );
    /* octave */
            Bconout ( 3, freq2 );
    /* casio default..temp */
            Bconout ( 3, 0 );
    /* freq playing */
            break;
        default:
            Bconout ( 3, 128 + chan2 + 16 );
            Bconout ( 3, freq2 );
    /* casio default..temp */
            Bconout ( 3, 64 );
    /* freq playing */
            break;
        }

    }

/*
**  There are eight voices permissible
**  with the AUTO-PLAYER, although
**  upping the number of voices to 16
**  would be a snap.  The variable oct
**  holds a value which tells the
**  AUTO-PLAYER if a note has been shut
**  off.  The function call noteoff turns
**  the MIDI note off.
*/

/*----- off voice 1-----*/

offv1()
    {
```

```
    int foff, addit;
    if ( note1 != 0 )
    {
    note1--;                    /* restore old value */
    foff = ftab1[ note1 ];      /* midi spec adjust */
    switch ( oct1 )
        {
        case 0x01:
            gfreq1 = -12;
            break;
        case 0x02:
            gfreq1 = 0;
            break;
        default:
            gfreq1 = 12;
            break;
        }
    noteoff ( 0, 0, foff + gfreq1 );
    /* turn note off */
    note1++;               /* restore new val */
    }
    }

/*----- off voice 2-----*/

offv2()
    {
    int foff, addit;
    if ( note2 != 0 )
    {
    note2--;                    /* restore old value */
    foff = ftab2[ note2 ];      /* midi spec adjust */
    switch ( oct2 )
        {
        case 0x01:
            gfreq2 = -12;
            break;
        case 0x02:
            gfreq2 = 0;
            break;
        default:
            gfreq2 = 12;
            break;
        }
    noteoff ( 0, 0, foff + gfreq2 ); /* turn note off */
    note2++;               /* restore new val */
    }
    }

/*----- off voice 3-----*/

offv3()
    {
    int foff, addit;     /* 200 */
    if ( note3 != 0 )
    {
    note3--;             /* restore old value */
    foff = ftab3[ note3 ];  /* midi spec adjust */
    switch ( oct3 )
        {
        case 0x01:
            gfreq3 = -12;
            break;
        case 0x02:
            gfreq3 = 0;
            break;
        default:
            gfreq3 = 12;
            break;
        }
    noteoff ( 1, 0, foff + gfreq3 );
    /* turn note off */
    note3++;             /* restore new val */
    }
    }

/*----- off voice 4-----*/

offv4()
    {
    int foff, addit;     /* 200 */
    if ( note4 != 0 )
    {
    note4--;             /* restore old value */
    foff = ftab4[ note4 ];  /* midi spec adjust */
    switch ( oct4 )
        {
        case 0x01:
            gfreq4 = -12;
            break;
        case 0x02:
            gfreq4 = 0;
            break;
        default:
            gfreq4 = 12;
            break;
        }
    noteoff ( 1, 0, foff + gfreq4 );
    /* turn note off */
    note4++;            /* restore new val */
    }
    }

/*----- off voice 5-----*/

offv5()
    {
    int foff, addit;    /* 200 */
    if ( note5 != 0 )
    {
    note5--;            /* restore old value */
    foff = ftab5[ note5 ];  /* midi spec adjust */
    switch ( oct5 )
        {
        case 0x01:
            gfreq5 = -12;
            break;
        case 0x02:
            gfreq5 = 0;
            break;
        default:
            gfreq5 = 12;
            break;
        }
    noteoff ( 2, 0, foff + gfreq5 ); /* turn note off */
    note5++;            /* restore new val */
    }
    }

/*----- off voice 6-----*/

offv6()
    {
    int foff, addit;    /* 200 */
    if ( note6 != 0 )
    {
    note6--;            /* restore old value */
    foff = ftab6[ note6 ];  /* midi spec adjust */
    switch ( oct6 )
        {
        case 0x01:
            gfreq6 = -12;
            break;
        case 0x02:
            gfreq6 = 0;
            break;
        default:
            gfreq6 = 12;
            break;
        }
    noteoff ( 2, 0, foff + gfreq6 ); /* turn note off */
    note6++;              /* restore new val */
    }
    }

/*----- off voice 7-----*/

offv7()
    {
    int foff, addit;      /* 200 */
    if ( note7 != 0 )
    {
    note7--;              /* restore old value */
    foff = ftab7[ note7 ];  /* midi spec adjust */
    switch ( oct7 )
        {
        case 0x01:
            gfreq7 = -12;
            break;
        case 0x02:
            gfreq7 = 0;
            break;
        default:
            gfreq7 = 12;
            break;
        }
    noteoff ( 3, 0, foff + gfreq7 ); /* turn note off */
    note7++;
    }           /* restore new val */
    }

/*----- off voice 8-----*/

offv8()
    {
    int foff, addit;      /* 200 */
    if ( note8 != 0 )
    {
```

```
    note8--;              /* restore old value */
    foff = ftab8[ note8 ];  /* midi spec adjust */
    switch ( oct8 )
        {
        case 0x01:
            gfreq8 = -12;
            break;
        case 0x02:
            gfreq8 = 0;
            break;
        default:
            gfreq8 = 12;
            break;
        }
    noteoff ( 3, 0, foff + gfreq8 ); /* turn note off */
    note8++;              /* restore new val */
    }
    }

/*
**  These routines turn the voices
**  on.  The variable fon will hold a
**  number which will holds the note
**  value.  If the note value is a 1
**  then the note is turned off and a
**  rest is simulated.
*/

/*----- on voice 1-----*/

onv1()
    {
    int fon, addit;
    fon = ftab1[ note1 ];   /* midi spec adjust */
    switch ( oct1 )
        {
        case 0x01:
            gfreq1 = -12;
            break;
        case 0x02:
            gfreq1 = 0;
            break;
        default:
            gfreq1 = 12;
            break;
        }
    switch ( fon )
        {
        case 0x01:
            noteoff ( 0, 0, fon + gfreq1 );
            break;
        default:
            noteon ( 0, 64, fon + gfreq1 );

            break;
        }
    }

/*----- off voice 2-----*/

onv2()
    {
    int fon, addit;
    fon = ftab2[ note2 ];    /* midi spec adjust */
    switch ( oct2 )
        {
        case 0x01:
            gfreq2 = -12;
            break;
        case 0x02:
            gfreq2 = 0;
            break;
        default:
            gfreq2 = 12;
            break;
        }
    switch ( fon )
        {
        case 0x01:
            noteoff ( 0, 0, fon + gfreq2 );
            break;
        default:
            noteon ( 0, 64, fon + gfreq2 );

            break;
        }
    }

/*----- on voice 3-----*/

onv3()
    {
    int fon, addit;
    fon = ftab3[ note3 ];    /* midi spec adjust */
    switch ( oct3 )
        {
        case 0x01:
            gfreq3 = -12;
            break;
        case 0x02:
            gfreq3 = 0;
            break;
        default:
            gfreq3 = 12;
            break;
        }
    switch ( fon )
        {
        case 0x01:
            noteoff ( 1, 0, fon + gfreq3 );
            break;
        default:
            noteon ( 1, 64, fon + gfreq3 );
            break;
        }
    }

/*----- on voice 4-----*/

onv4()
    {
    int fon, addit;
    fon = ftab4[ note4 ];    /* midi spec adjust */
    switch ( oct4 )
        {
        case 0x01:
            gfreq4 = -12;
            break;
        case 0x02:
            gfreq4 = 0;
            break;
        default:
            gfreq4 = 12;
            break;
        }
    switch ( fon )
        {
        case 0x01:
            noteoff ( 1, 0, fon + gfreq4 );
            break;
        default:
            noteon ( 1, 64, fon + gfreq4 );
            break;
        }
    }

/*----- on voice 5-----*/

onv5()
    {
    int fon, addit;
    fon = ftab5[ note5 ];    /* midi spec adjust */
    switch ( oct5 )
        {
        case 0x01:
            gfreq5 = -12;
            break;
        case 0x02:
            gfreq5 = 0;
            break;
        default:
            gfreq5 = 12;
            break;
        }
    switch ( fon )
        {
        case 0x01:
            noteoff ( 2, 0, fon + gfreq5 );
            break;
        default:
            noteon ( 2, 64, fon + gfreq5 );
            break;
        }
    }

/*----- on voice 6-----*/

onv6()
    {
    int fon, addit;
    fon = ftab6[ note6 ];    /* midi spec adjust */
    switch ( oct6 )
        {
        case 0x01:
            gfreq6 = -12;
            break;
        case 0x02:
            gfreq6 = 0;
            break;
        default:
            gfreq6 = 12;
            break;
        }
    switch ( fon )
        {
        case 0x01:
            noteoff ( 2, 0, fon + gfreq6 );
            break;
        default:
            noteon ( 2, 64, fon + gfreq6 );
            break;
        }
    }

/*----- on voice 7-----*/

onv7()
    {
    int fon, addit;
    fon = ftab7[ note7 ];   /* midi spec adjust */
    switch ( oct7 )
        {
        case 0x01:
            gfreq7 = -12;
            break;
        case 0x02:
            gfreq7 = 0;
            break;
        default:
            gfreq7 = 12;
            break;
        }
    switch ( fon )
        {
        case 0x01:
            noteoff ( 3, 0, fon + gfreq7 );
            break;
        default:
            noteon ( 3, 64, fon + gfreq7 );
            break;
        }
    }
```

```
/*----- on voice 8-----*/

onv8()
    {
    int fon, addit;
    fon = ftab8[ note8 ];    /* midi spec adjust */
    switch ( oct8 )
        {
        case 0x01:
            gfreq8 = -12;
            break;
        case 0x02:
            gfreq8 = 0;
            break;
        default:
            gfreq8 = 12;
            break;
        }
    switch ( fon )
        {
        case 0x01:
            noteoff ( 3, 0, fon + gfreq8 );
            break;
        default:
            noteon ( 3, 64, fon + gfreq8 );
            break;
        }
    }

/*
**  This routine is the heart of the colsol
**  speaker play routine.  When it is called the
**  routine receives three parameters.  The sound
**  chip inside the Atari ST uses has three voices.
**  The variable vce is the voice number. The
**  variable frq is the frequency number and
**  vme is the volume.  This rouine may be lifted
**  and used in your own programs.
*/

/*-----this routine plays a voice (vce 1-3), frequency (frq 1-96),----*/
/*-----and volume (vme 0-15)------------------------------------------*/

playv ( vce, frq, vme )
    int vce, frq, vme;
    {
    vme = 12;
    if ( frq > 24 )      /* adjust for octave fix */
        frq -= 24;  /* dumb dumb */
    switch ( vce )
        {
        case 0x01:
            if ( frq == 1 )      /* ok ok idiot fix */
                Giaccess ( 0,0x88 ); /* dumb dumb dumb  */
            else
                Giaccess ( vme, 0x88 );
            Giaccess ( ftune[frq], 0x80 );
            Giaccess ( ctune[frq], 0x81 );
            break;
        case 0x02:
            if ( frq == 1 )      /* ok ok idiot fix */
                Giaccess ( 0,0x89 ); /* dumb dumb dumb  */
            else                 /* 200 */
                Giaccess ( vme, 0x89 );
            Giaccess ( ftune[frq], 0x82 );
            Giaccess ( ctune[frq], 0x83 );
            break;
        case 0x03:
            if ( frq == 1 )      /* ok ok idiot fix */
                Giaccess ( 0,0x8a ); /* dumb dumb dumb  */
            else
                Giaccess ( vme, 0x8a );
            Giaccess ( ftune[frq], 0x84 );
            Giaccess ( ctune[frq], 0x85 );
            break;
        }
    }

/*
**  This is the Grand-daddy routine of the
**  AUTO-PLAYER.  The logic is as follows:
**      1- get a duration value for voice 1
**      2- is it 0?
**      3- if yes - get new note frequency
**         and duration..if no - let play
**      .
**      .
**      .
**      4- very short wait
**      5- increment measure counter
**      6- update player screen
**      7- check mouse & control buttons and key
**      8- continue loop until done
*/
```

```
    /*------this routine will play the notes for the demo music------*/

playmus()
    {
    gpflag = 0;
    exflag = 0;
    while ( exflag == 0 )
        {
        if ( dur1 == 0 )
            getv1();

        if ( dur2 == 0 )
            getv2();

        if ( dur3 == 0 )
            getv3();

        if ( dur4 == 0 )
            getv4();

        if ( dur5 == 0 )
            getv5();

        if ( dur6 == 0 )
            getv6();

        if ( dur7 == 0 )
            getv7();

        if ( dur8 == 0 )
            getv8();

        delay1( timex, timey ); /* sets tempo */

            dur1--;
            dur2--;
            dur3--;
            dur4--;
            dur5--;
            dur6--;
            dur7--;
            dur8--;

        first = 1;
        lmeas();
        progdem();
        update();
        if ( gpflag == 255 )     /* tune over exit */
            exflag = 1;
        graf_mkstate( &gr_mkmx, &gr_mkmy, &gr_mkmstate, &gr_mkkstate );
        if ( gr_mkkstate == 4 ) /* control */
            exflag = 1;
        }
    }

/*
**  This youtine calculates the measure
**  for the AUTO-PLAYER.  The numbers before
**  the time signatures repsent player loop
**  numbers.  These times are NOT etched in
**  stone and may change in future upgrades
**  of ST MUSIC BOX FILES.  If you want to know
**  the current conversion values for the latest
**  version of ST MUSIC BOX, write Len or Dennis
**  at ABACUS and they will send you the newest
**  file information.
*/

/*-----calculate the measure-----*/

/*
**  16=2/4, 24=3/4, 32=4/4, 40=5/4
*/

lmeas()
    {
    cnt32++;
    if ( cnt32 == time )
        {
        cnt32 = 0;
        meascnt++;
        }
    }

/*
**  This is the routine which updates
**  the program change information
**  every measure.  It also
**  calls a routine to update the
**  screen.  This routine has to be
**  VERY fast or it would give the
**  listner a very slight music retard
**  at the beginning of each measure.
**  Of course, that would be unacceptable.
*/

/*-----demo of program change----*/


/*
**  16=2/4, 24=3/4, 32=4/4, 40=5/4
*/

progdem()
    {
    int ttemp;
    ttemp = time;
    progcnt++;
    if ( progcnt == time )
        {
            progcnt = 0;
            chancnt = prog0c[ meascnt - 1 ];
            gprog = chancnt + 1;
            chprog ( 0, chancnt );
            chancnt = prog1c[ meascnt - 1 ];
            gprog1 = chancnt + 1;
            chprog ( 1, chancnt );
            chancnt = prog2c[ meascnt - 1 ];
            gprog2 = chancnt + 1;
            chprog ( 2, chancnt );
            chancnt = prog3c[ meascnt - 1 ];
            gprog3 = chancnt + 1;
            chprog ( 3, chancnt );
            timey = ldtmpo[ meascnt - 1 ];
        }
    }

/*
**  This routine grabs the notes from
**  the frequency buffers and duration
**  values from the duration buffers.
**  The function playv activates the
**  consol speaker, onv1 turns on the MIDI
**  and movell updates the position of the
**  dancing notes on the screen.  Note that
**  the MIDI note must be turned OFF before
**  another is turned on.  That's just the
**  way it's gotta be in order for the music
**  to be clean of different machines.
*/
```

```
 /*-----get new notes for the voices-----*/

getv1()
    {
    if ( dtab1[ note1 ] == 0 )
        note1++;
    dur1 = dtab1[ note1 ];
    if ( ftab1[ note1] == 255 || ftab1[ note1 ] == 0 || vc1 == 0 )
        {
        gpflag |= 1;
        offv1();
        playv( 1, 1, 0 );
        move11( 1 );
        }
    else
        {
        playv ( 1, ftab1[ note1 ], 8 );
        offv1();
        move11 ( ftab1[ note1 ] );/* print to screen */
        onv1();                  /* play midi */
        note1++;
        }
    }

getv2()
    {
    int temp2;
    if ( dtab2[ note2 ] == 0 )
        note2++;
    dur2 = dtab2[ note2 ];
    if ( ftab2[ note2 ] == 255 || ftab2[ note2 ] == 0 || vc2 == 0 )
        {
        gpflag |= 2;
        offv2();
        move21( 1 );
        }
    else
        {
        offv2();
        move21 ( ftab2[ note2 ] ) ;/* print to screen */
        onv2();              /* play midi */
        note2++;
        }
    }

getv3()
    {
```

```
    if ( dtab3[ note3 ] == 0 )
        note3++;
    dur3 = dtab3[ note3 ];
    if ( ftab3[ note3 ] == 255 || ftab3[ note3 ] == 0 || vc3 == 0 )
        {
        gpflag |= 4;
        offv3();
        playv( 2, 1, 0 );
        move3l( 1 );
        }
    else
        {
        playv ( 2, ftab3[ note3 ], 8 );
        offv3();
        move3l ( ftab3[ note3 ] ) ;/* print to screen */
        onv3();                    /* play midi */
        note3++;
        }
    }

getv4()
    {
    if ( dtab4[ note4 ] == 0 )
        note4++;
    dur4 = dtab4[ note4 ];
    if ( ftab4[ note4 ] == 255 || ftab4[ note4 ] == 0 || vc4 == 0 )
        {
        gpflag |= 8;
        offv4();
        move4l( 1 );
        }
    else
        {
        offv4();
        move4l ( ftab4[ note4 ] ) ; /* print to screen */
        onv4();                /* play midi */
        note4++;
        }
    }

getv5()
    {
    if ( dtab5[ note5 ] == 0 )
        note5++;
    dur5 = dtab5[ note5 ];
    if ( ftab5[ note5 ] == 255 || ftab5[ note5 ] == 0 || vc5 == 0 )
        {
          gpflag |= 16;
          offv5();
          playv( 3, 1, 0 );
          move5l( 1 );
          }
     else
          {
          playv ( 3, ftab5[ note5 ], 8 );
          offv5();
          move5l ( ftab5[ note5 ] ) ; /* print to screen */
          onv5();                /* play midi */
          note5++;
          }
     }

getv6()
     {
     if ( dtab6[ note6 ] == 0 )
          note6++;
     dur6 = dtab6[ note6 ];
     if ( ftab6[ note6 ] == 255 || ftab6[ note6 ] == 0 || vc6 == 0 )
          {
          gpflag |= 32;
          offv6();
          move6l( 1 );
          }
     else
          {
          offv6();
          move6l ( ftab6[ note6 ] ) ; /* print to screen */
          onv6();                /* play midi */
          note6++;
          }
     }

getv7()
     {
     if ( dtab7[ note7 ] == 0 )
          note7++;
     dur7 = dtab7[ note7 ];
     if ( ftab7[ note7 ] == 255 || ftab7[ note7 ] == 0 || vc7 == 0 )
          {
          gpflag |= 64;
          offv7();
          move7l( 1 );
          }
     else
```

```
            {
            offv7();
            move7l ( ftab7[ note7 ] ) ; /* print to screen */
            onv7();                /* play midi */
            note7++;
            }
        }

getv8()
        {
        if ( dtab8[ note8 ] == 0 )
            note8++;
        dur8 = dtab8[ note8 ];
        if ( ftab8[ note8 ] == 255 || ftab8[ note8 ] == 0 || vc8 == 0 )
            {
            gpflag |= 128;
            offv8();
            move8l( 1 );
            }
        else
            {
            offv8();
            move8l ( ftab8[ note8 ] ); /* print to screen */
            onv8();                /* play midi */
            note8++;
            }
        }

/*
**  This is just a very simple delay which will
**  determine the tempo of the music.  A larger delay
**  ill mean a slower tempo.
*/

/*------delay which sets the tempo of the music--------*/

delay1 ( d1, d2 )
        int d1, d2;
        {
        int d3, d4;
        for ( d3 = 0; d3 < d1; d3++ )
            {
            for ( d4 = 0; d4 < d2; d4++ )
            }
        }
```

```
/*
*   reserved for future use
*/

ldshow()
{
    }

/*
**  end of mdrive.c file
*
```

4.4 `sets.c` source code

This file is rich, containing a variety of useful subroutines. Many of these routines are NOT functional in the AUTO-PLAYER and are used with the full ST MUSIC BOX program. They were left in the file to help us cope with the extraordinary time pressure we were under in order to meet our COMDEX demonstration deadline and we felt you might find the listing of these routines helpful in your own programming.

```
/*  sets.c      */

/*
**  include the definition libraries here
*/

#include    <a:portab.h>
#include    <a:machine.h>
#include    <a:obdefs.h>
#include    <a:define.h>
#include    <a:gemdefs.h>
#include    <a:osbind.h>

/*  externs     */

extern  int wbox, hbox, dur1, dur2, dur3, port_state;
extern  int contrl[12], intin[256], ptsin[256], ptsout[256];
extern  int intin[256], intout[256],pxy[10];
extern  int newbyte[8], oldbyte[8], typedone, hbox1, wbox1, hchar1,
wchar1;
extern  int l_intin[20], l_out[128], l_ptsin[20];
extern  int kreturn, i, iter;
extern  int ev_kreturn,gr_mkmx,gr_mkmy,groption,stylem;
extern  int endm,mode,set_mode;
extern  int set_font,handle,dummy,gr_mkmstate,gr_mkkstate,prtmem;
extern  int holdrez,set_effect;
extern  int holdx,holdy,whesc,whesc1,color_index,colorm,holdclr;
extern  int gr_mkresvd,intstyle,topbot,height,width,exflag;
extern  int widd,mull,radius;
extern  int begang,endang,yradius,x_boundary,y_boundary,k,k1,k2,k3,k4;
extern  int n,u,ab,kk,row,count,column,sp,mod,byte,j,quot,rem,demo;
extern  int note1, note2, note3;
extern  int scrold, scrnew, mult, ymult, findex, fcolor,;
extern  int linev, lineh, linesw;
extern   int byte1, byte2, ig, sgflag,prtsw,path;
extern   int timex, timey, first, octup, octsw0, octsw1, octsw2;
extern   int meascnt, cnt32;
extern   int chancnt, progcnt, gfreq1, gfreq2, gfreq3;
extern   long *ptr1, *ptr2, *ptr3, *ptr4, *ptr5, *ptr6;
extern   int gpsw, gptime, dblo1, gvibsw, gprog;
extern   char tmpo_buffer[ 264 ], c;
extern   long logicbase,screen1,screen2,orgscrn,ltemp;
extern   char buffer1[32767],xfile_name[64]; /* main display screen */
char     ystr[2] = { '\0', '\0' };

/*
*   globals in this file
*/

int gltemp;
int respold = 2;

/*
*   definitions
*/

#define      lgdval  2    /*  2 for col & 1 for mono */

/*
**   sets the correct key for rest sequence
*/

/*
**   when the editor was in its early stages we were using
**   the keyboard to enter notes.  Rather that have the
**   user interface remember the key codes needed to enter
**   music data we decided to use simple numbers instead.
**   What this routine does is it takes the number parameter
**   calculated by the mouse assessment routine and sets
**   a global variable 'kreturn' as if the correct key had
**   been pressed.
*/


ldsetr()
     {
     switch ( respold )
          {
          case 0x00:
               kreturn = 0x0837;
               break;
```

```
        case 0x01:
            kreturn = 0x0938;
            break;
        case 0x02:
            kreturn = 0x0a39;
            break;
        case 0x03:
            kreturn = 0x0b30;
            break;
        case 0x04:
            kreturn = 0x0c2d;
            break;
        case 0x05:
            kreturn = 0x0d3d;
            break;
        }
    }

/*
*   piano color key routines
*/


/*
**  This function first turns OFF the color of the last piano
**  key which had been pressed and then colors the new key.
*/


ldgetk()
    {
    ldoffkey(); /* turn key off */
    ldonkey();  /* turn key on */
    }

/*
*   turn key off to bkgd color
*/

/*
**  We use the VDI function contourfill to color the keys.
*/


ldoffkey()
    {
    int isy, isx, dumdum;
    v_hide_c( handle );
    isy = ( 298 + 16 ) / lgdval;
    dumdum = vsf_interior( handle, 1 );
    dumdum = vswr_mode( handle, 1 );
    dumdum = vsf_style( handle, 1 );
    dumdum = vsf_color( handle, 0 );
    switch ( gltemp )
        {
        case 0x00:
            isx = 315;
            v_contourfill( handle, isx, isy, 1 );
            break;
        case 0x01:
            isx = 345;
            v_contourfill( handle, isx, isy, 1 );
            break;
        case 0x02:
            isx = 368;
            v_contourfill( handle, isx, isy, 1 );
            break;
        case 0x03:
            isx = 380;
            v_contourfill( handle, isx, isy, 1 );
            break;
        case 0x04:
            isx = 410;
            v_contourfill( handle, isx, isy, 1 );
            break;
        case 0x05:
            isx = 440;
            v_contourfill( handle, isx, isy, 1 );
            break;
        case 0x06:
            isx = 470;
            v_contourfill( handle, isx, isy, 1 );
            break;
        case 0x07:
            isx = 500;
            v_contourfill( handle, isx, isy, 1 );
            break;
        case 0x08:
            isx = 515;
            v_contourfill( handle, isx, isy, 1 );
            break;
        case 0x09:
            isx = 550;
            v_contourfill( handle, isx, isy, 1 );
```

```
          break;
      case 0x0a:
          isx = 560;
          v_contourfill( handle, isx, isy, 1 );
          break;
      case 0x0b:
          isx = 590;
          v_contourfill( handle, isx, isy, 1 );
          break;
      }
  v_show_c( handle, 1 );
  }

/*
*   turn on key
*/

ldonkey()
  {
  int isy, isx, dumdum;
  v_hide_c( handle );
  isy = ( 298 + 16 ) / lgdval;
  dumdum = vsf_interior( handle, 1 );
  dumdum = vswr_mode( handle, 1 );
  dumdum = vsf_style( handle, 1 );
  dumdum = vsf_color( handle, 2 );
  bing1();          /* try 2 */
  switch ( kreturn )
      {
      case 0x2c5a:          /* c */
      case 0x2c7a:
          gltemp = 0;
          isx = 315;
          v_contourfill( handle, isx, isy, 1 );
          break;
      case 0x1f53:          /* c# */
      case 0x1f73:
          gltemp =1;
          isx = 345;
          v_contourfill( handle, isx, isy, 1 );
          break;
      case 0x2d58:          /* d */
      case 0x2d78:
          gltemp = 2;
          isx = 368;
          v_contourfill( handle, isx, isy, 1 );
          break;
```

```
    case 0x2044:          /* d# */
    case 0x2064:
        gltemp = 3;
        isx = 380;
        v_contourfill( handle, isx, isy, 1 );
        break;
    case 0x2e43:          /* e */
    case 0x2e63:
        gltemp = 4;
        isx = 410;
        v_contourfill( handle, isx, isy, 1 );
        break;
    case 0x2f56:          /* f */
    case 0x2f76:
        gltemp = 5;
        isx = 440;
        v_contourfill( handle, isx, isy, 1 );
        break;
    case 0x2247:          /* f# */
    case 0x2267:
        gltemp = 6;
        isx = 470;
        v_contourfill( handle, isx, isy, 1 );
        break;
    case 0x3042:          /* g */
    case 0x3062:
        gltemp = 7;
        isx = 500;
        v_contourfill( handle, isx, isy, 1 );
        break;
    case 0x2348:          /* g# */
    case 0x2368:
        gltemp = 8;
        isx = 515;
        v_contourfill( handle, isx, isy, 1 );
        break;
    case 0x314e:          /* a */
    case 0x316e:
        gltemp = 9;
        isx = 550;
        v_contourfill( handle, isx, isy, 1 );
        break;
    case 0x244a:          /* a# */
    case 0x246a:
        gltemp = 10;
        isx = 560;
        v_contourfill( handle, isx, isy, 1 );
```

```
            break;
        case 0x324d:          /* b */
        case 0x326d:
            gltemp = 11;
            isx = 590;
            v_contourfill( handle, isx, isy, 1 );
            break;
        }
    v_show_c( handle, 1 );
    }

/*
**  binger
*/

/*
**  Bing is really the wrong work hear.  Short RASP sound
**  would be a better description.
*/


bing1()
    {
    int     hf;
    port_state = Giaccess ( port_state, 0x07 );
    n = port_state & 0xf8;
    n |= 0x38;
    Giaccess (n, 0x87 );
    Giaccess ( 12, 0x88 );
    setenv ( 256, 1024, 3 );
    for ( hf = 96; hf < 1; hf-- )
        {
        playv( 1, hf, 12 );
        delay3();
        delay3();
        delay3();
        delay3();
        delay3();
        delay3();
        }
    Giaccess ( 0x00, 0x88 );
    Giaccess ( port_state, 0x87 );
    }

/*
**  Here is the LONGER RASP sound.
*/
```

```
bing2()
    {
    int     hf;
    port_state = Giaccess ( port_state, 0x07 );
    n = port_state & 0xf8;
    n |= 0x38;
    Giaccess (n, 0x87 );
    Giaccess ( 12, 0x88 );
    setenv ( 256, 1024, 3 );
    for ( hf = 96; hf < 1; hf-- )
        {
        playv( 1, hf, 12 );
        delay3();
        }
    Giaccess ( 0x00, 0x88 );
    Giaccess ( port_state, 0x87 );
    }

bing25()
    {
    int bctr;
    for ( bctr = 0; bctr < 100; bctr++ )
        bing2();
    }

/*
**  short delay
*/

delay3()
    {
    int d7, d8;
    for ( d7 = 0; d7 < ( 180 * 8 ); d7++ );
        for ( d8 = 0; d8 < 500; d8++ )
    }

/*
**  Map the mouse on key press
*/


/*
**  This routine would print the mouse x and mouse y
**  positions to the screen.  Note that the print3d()
**  call is the one which does the actual printing.
**  The two calls are REMed out because they were used
**  only in the early stages of programming the editor.
**  They were left here if needed for upgrades.
*/


mapmse()
    {
/*
**  print3d( gr_mkmx, 1, 1 );
**  print3d( gr_mkmy, 6, 1 );
*/
    bing25();
    bing25();
    bing25();
    bing25();
    }

/*
*   reads the mouse for note entry
*   what a pain - not so bad on 2nd thought
*/

/*
**  In many ways 'C' is such a wonderful programming
**  language!!  In order to check the x & y coordinate mouse
**  location you only need to check the UPPER LEFT and LOWER
**  RIGHT of the ICON in question.  See the
**  'alter tempo' section of this function.
**  You can see that the screen location check
**  has been reduced to one line of code.  Thank
**  you 'C'.
*/


ldsendk()
    {
    v_hide_c( handle );
    graf_mkstate( &gr_mkmx, &gr_mkmy, &gr_mkmstate, &gr_mkkstate );

    if ( gr_mkmy < 118 || gr_mkmy > 198 )
        {
        kreturn = 0;
        mapmse();        /* temp for mapping */
        }
/*
**  alter tempo
*/
```

```
    else if
    (gr_mkmx < 182 && gr_mkmx > 82 && gr_mkmy > 180 && gr_mkmy < 194 )
        {
        timey = gr_mkmx - 80;    /* fix for player */
        if ( timey >= 100 ) /* keep to under 100 */
            timey = 99;
        tmpo_buffer[0] = tochar( timey );
        ldstemp();        /* print tempo number */
        kreturn = 0;
        }
/*
**  access i/o disk
*/
    else if
    (gr_mkmx < 27 && gr_mkmx > 5 && gr_mkmy > 120 && gr_mkmy < 137 )
        kreturn = 0x4400;    /* f10 */


/*
**  access play routine
*/
    else if
    (gr_mkmx > 17 && gr_mkmx < 61 && gr_mkmy > 149 && gr_mkmy < 165 )
        kreturn = 0x4000;    /* f6 */


/*
**  access midi window
*/

    else if
    ( gr_mkmx > 80 && gr_mkmx < 100 && gr_mkmy > 160 && gr_mkmy < 172 )
        kreturn = 0x3f00;    /* f5 */
/*
**  set the rest
*/


    else if
    ( gr_mkmx > 240 && gr_mkmx < 267 && gr_mkmy > 181 && gr_mkmy < 187 )
        ldsetr();


/*
**  select a dotted note
*/


    else if
    ( gr_mkmx > 134 && gr_mkmx < 155 && gr_mkmy > 160 && gr_mkmy < 169 )
        {
        bing25();
        kreturn = 0x342e;    /* . */
        }
/*
**  change sharp and flat
*/
    else if
    (gr_mkmx > 80 && gr_mkmx < 100 && gr_mkmy > 145 && gr_mkmy < 153 )
        kreturn = 0x1e61;    /* a */

/*
**  hit return key
*/
    else if
    (gr_mkmx > 138 && gr_mkmx < 156 && gr_mkmy > 145 && gr_mkmy < 153 )
        kreturn = 0x1c0d;    /* return */

/*
**  backspace
*/
    else if
    (gr_mkmx > 192 && gr_mkmx < 213 && gr_mkmy > 145 && gr_mkmy < 154 )
        kreturn = 0x0e08;    /* backspace */

/*
**  whole note
*/
    else if
    (gr_mkmx > 241 && gr_mkmx <253 && gr_mkmy > 145 && gr_mkmy < 153 )
        {
        kreturn = 0x0231;    /* 1 */
        respold = 0;
        }
/*
**  half note
*/
    else if
    (gr_mkmx > 256 && gr_mkmx < 267 && gr_mkmy > 145 && gr_mkmy < 153 )
        {
        kreturn = 0x0332;    /* 2 */
        respold = 1;
        }
/*
**  quarter note
*/
    else if
    ( gr_mkmx > 254 && gr_mkmx < 267 && gr_mkmy > 154 && gr_mkmy < 165 )
        {
         kreturn = 0x0433;    /* 3 */
         respold = 2;
         }
/*
**   eighth note
*/
     else if
     (gr_mkmx > 254 && gr_mkmx < 267 && gr_mkmy > 167 && gr_mkmy < 177 )
         {
         kreturn = 0x0534;    /* 4 */
         respold = 3;
         }
/*
**   sixteenth note
*/
     else if
     (gr_mkmx > 241 && gr_mkmx < 253 && gr_mkmy > 155 && gr_mkmy < 164 )
         {
         kreturn = 0x0635;    /* 5 */
         respold = 4;
         }
/*
**   thirty second
*/
     else if
     (gr_mkmx > 241 && gr_mkmx < 253 && gr_mkmy > 167 && gr_mkmy < 179 )
         {
         kreturn = 0x0736;    /* 6 */
         respold = 5;
         }
/*
**   change measure - down arrow key
*/
     else if
     (gr_mkmx > 68 && gr_mkmx < 86 && gr_mkmy > 121 && gr_mkmy < 129 )
         kreturn = 0x5000;

/*
**   change measure - up arrow key
*/
     else if
     (gr_mkmx > 94 && gr_mkmx < 111 && gr_mkmy > 120 && gr_mkmy < 130 )
         kreturn = 0x4800;

/*
**   change voice down -  [<] key
*/
```

```
    else if
    (gr_mkmx > 123 && gr_mkmx < 141 && gr_mkmy > 121 && gr_mkmy < 130 )
        kreturn = 0x333c;

/*
**  change voice up - [>] key
*/
    else if
    (gr_mkmx > 150 && gr_mkmx < 169 && gr_mkmy > 121 && gr_mkmy < 130 )
        kreturn = 0x343e;

/*
**  change octave down - left arrow key
*/
    else if
    (gr_mkmx > 179 && gr_mkmx < 195 && gr_mkmy > 121 && gr_mkmy < 131 )
        kreturn = 0x4b00;

/*
**  change octave up - right arrow key
*/
    else if
    (gr_mkmx > 205 && gr_mkmx < 224 && gr_mkmy > 121 && gr_mkmy < 131 )
        kreturn = 0x4d00;

/*
**  read the drawn keyboard to pass note data
*/
    else if ( gr_mkmx < 300 || gr_mkmx > 615 )
        {
        kreturn = 0;
        mapmse();          /* temp for mapping */
        }
    else if ( gr_mkmy > 158 )          /* hopefully */
        tckbot();              /* bottom key check */
    else
        tcktop();              /* top key check */
    v_show_c( handle, 1 );
    }

/*
*   short delay
*/

delay2()
    {
    int d3, d4;
    for ( d3 = 0; d3 < 300; d3++ )
        for ( d4 = 0; d4 < 100; d4++ )
    }

/*
*   bottom key mouse entry routine
*/


/*
**  Here we used a slightly different logic to assess which
**  key the mouse pointer was over.  Comparing the methods:
**  One is six and the other is half a dozen.
**
*/


tckbot()
    {
    if ( gr_mkmx < 345 )
        ldkk1();
    else if ( gr_mkmx < 390 )
        ldkk3();
    else if ( gr_mkmx < 435 )
        ldkk5();
    else if ( gr_mkmx < 480 )
        ldkk6();
    else if ( gr_mkmx < 525 )
        ldkk8();
    else if ( gr_mkmx < 570 )
        ldkk10();
    else
        ldkk12();
    }

/*
*   top key mouse entry routine
*/

tcktop()
    {
    if ( gr_mkmx < 330 )
        ldkk1();
    else if ( gr_mkmx < 360 )
        ldkk2();
    else if ( gr_mkmx < 375 )
        ldkk3();
    else if ( gr_mkmx < 405 )
        ldkk4();
    else if ( gr_mkmx < 435 )
        ldkk5();
    else if ( gr_mkmx < 465 )
        ldkk6();
    else if ( gr_mkmx < 495 )
        ldkk7();
    else if ( gr_mkmx < 510 )
        ldkk8();
    else if ( gr_mkmx < 540 )
        ldkk9();
    else if ( gr_mkmx < 555 )
        ldkk10();
    else if ( gr_mkmx < 585 )
        ldkk11();
    else
        ldkk12();
    }

/*
*   simulated mouse-keybd strokes
*/

ldkk1()
    {
    ldoffkey();
    kreturn = 0x2c5a;
    ldonkey();
    }

ldkk2()
    {
    ldoffkey();
    kreturn = 0x1f53;
    ldonkey();
    }

ldkk3()
    {
    ldoffkey();
    kreturn = 0x2d58;
    ldonkey();
    }

ldkk4()
    {
     ldoffkey();
     kreturn = 0x2044;
     ldonkey();
     }

ldkk5()
     {
     ldoffkey();
     kreturn = 0x2e43;
     ldonkey();
     }

ldkk6()
     {
     ldoffkey();
     kreturn = 0x2f56;
     ldonkey();
     }

ldkk7()
     {
     ldoffkey();
     kreturn = 0x2247;
     ldonkey();
     }

ldkk8()
     {
     ldoffkey();
     kreturn = 0x3042;
     ldonkey();
     }

ldkk9()
     {
     ldoffkey();
     kreturn = 0x2348;
     ldonkey();
     }

ldkk10()
     {
     ldoffkey();
     kreturn = 0x314e;
     ldonkey();
     }
```

```
ldkk11()
    {
    ldoffkey();
    kreturn = 0x244a;
    ldonkey();
    }

ldkk12()
    {
    ldoffkey();
    kreturn = 0x324d;
    ldonkey();
    }

/*
**  Here is a useful little function which will print a
**  two digit number to the screen.  We use it all the
**  time.  The values passed are:
**  1) variable ..value..
**  2) x cursor position where number will print
**  3) y number location where number will print
**  Note the use of register variables.  Use of
**  register variables in 'C' will speed up the code
**  sometimes by a factor of three.  Since the print
**  screen routines were used in the middle of the
**  actual playing we tried to write quick code at
**  all costs.
*/


int     print2d ( val, xloc, yloc )
register    int val, xloc, yloc;
    {
    char    dbuff[4];
    dbuff[0] = '\0';
    dbuff[1] = '\0';
    dbuff[0] = val / 10;
    dbuff[0] += 48;
    v_gtext(handle,width*xloc,height*yloc,dbuff);
    dbuff[0] = val % 10;
    dbuff[0] += 48;
    ++xloc;
    v_gtext(handle,width*xloc,height*yloc,dbuff);
    }
```

```
/*--print a 3 digit number--call value,xloc,yloc--*/

int         print3d ( val3, xloc3, yloc3 )
register    int val3, xloc3, yloc3;
    {
    int temp3;
    char    dbuff[4];
    dbuff[0] = '\0';
    dbuff[1] = '\0';
    dbuff[0] = val3 / 100;
    temp3 = val3 % 100;
    dbuff[0] += 48;
    v_gtext(handle,width*xloc3,height*yloc3,dbuff);
    dbuff[0] = temp3 / 10;
    dbuff[0] += 48;
    ++xloc3;
    v_gtext(handle,width*xloc3,height*yloc3,dbuff);
    dbuff[0] = temp3 % 10;
    dbuff[0] += 48;
    ++xloc3;
    v_gtext(handle,width*xloc3,height*yloc3,dbuff);
    }


/*
**  This function call prints the screen for the AUTO-PLAYER.
**
*/


prtdemo()
    {

    int ilen,xgrid, ygrid, dline;

/*
**  first we print the numbers of the voices
*/


        v_gtext ( handle, width*62,height*10,"# 1 ");
        v_gtext ( handle, width*62,height*12,"# 2 ");
        v_gtext ( handle, width*62,height*14,"# 3 ");
        v_gtext ( handle, width*62,height*16,"# 4 ");
        v_gtext ( handle, width*62,height*18,"# 5 ");
        v_gtext ( handle, width*62,height*20,"# 6 ");
        v_gtext ( handle, width*62,height*22,"# 7 ");
        v_gtext ( handle, width*62,height*24,"# 8 ");

/*
**  Print the TITLE and our NAMES (- for vanity, of course!)
**
*/

    v_gtext( handle,width*2,height*2,
"'ST MUSIC BOX' AUTOPLAYER                       Filename ");
        v_gtext ( handle, width*2,height*3,
"by Len Dorfman and Dennis Young    (c) 1986");
        v_gtext(handle,width*2, height*4,
"Pressing CONTROL will abort the song and autoload a new one.");
        v_gtext(handle,width*2, height*1,
"From the ABACUS book ATARI ST Introduction to MIDI Programming");

/*
* print the filename
*/

    for(dline = 50,ilen = 2; ilen < 15; ilen++,dline++ )
        {
        ystr[0] = xfile_name[ilen];
        v_gtext(handle,width*dline,height*3,ystr);
        }

/*
**  Print the MIDI and play parameter labels\
*/


    v_gtext(handle,width*2,height*6,
"PORTAMENTO SW          PORTAMENTO RATE          TEMPO        VOICE
MEASURE");
    v_gtext(handle,width*2,height*7,
"CNTRL CD 1 CH 0        CNTRL CD 1 CH 1                       NUMBER
NUMBER ");
    v_gtext(handle,width*2,height*8,
"CNTRL CD 1 CH 2        CNTRL CD 1 CH 3           ");
    v_gtext(handle,width*71,height*10,"CHAN #1");
    v_gtext(handle,width*71,height*12,"PR. #  ");
    v_gtext(handle,width*71,height*14,"CHAN #2");
    v_gtext(handle,width*71,height*16,"PR. #  ");
    v_gtext(handle,width*71,height*18,"CHAN #3");
    v_gtext(handle,width*71,height*20,"PR. #  ");
    v_gtext(handle,width*71,height*22,"CHAN #4");
    v_gtext(handle,width*71,height*24,"PR. #  ");
```

```
/*
**  Print the boundaries for the little animated shapes
**  which dance about the screen.
*/


/* print vertical lines */

    for (dline = 0, xgrid = 480; dline < 2; dline++,xgrid += 72)
        {
        pxy[0] = xgrid;
        pxy[1] = 39;
        pxy[2] = xgrid;
        pxy[3] = 199;
        v_pline(handle,2,pxy);
        }

    for (dline = 0, xgrid = 560; dline < 2; dline++,xgrid += 72)
        {
        pxy[0] = xgrid;
        pxy[1] = 39;
        pxy[2] = xgrid;
        pxy[3] = 199;
        v_pline(handle,2,pxy);
        }

    for (dline = 0, xgrid = 0; dline < 2; dline++,xgrid += 480)
        {
        pxy[0] = xgrid;
        pxy[1] = 39;
        pxy[2] = xgrid;
        pxy[3] = 66;
        v_pline(handle,2,pxy);
        }

    for (dline = 0, xgrid = 0; dline < 2; dline++,xgrid += 480)
        {
        pxy[0] = xgrid;
        pxy[1] = 71;
        pxy[2] = xgrid;
        pxy[3] = 71 + ( 8 * 16 );
        v_pline(handle,2,pxy);
        }


/* print horizontal lines in grid */
```

```
    for (dline = 0, xgrid = 71; dline < 9; dline++, xgrid = xgrid + 16)
        {
        pxy[0] = 0;
        pxy[1] = xgrid;
        pxy[2] = 480;
        pxy[3] = xgrid;
        v_pline(handle,2,pxy);
        }

    for (dline = 0, xgrid = 39; dline < 2; dline++, xgrid += 28)
        {
        pxy[0] = 0;
        pxy[1] = xgrid;
        pxy[2] = 480;
        pxy[3] = xgrid;
        v_pline(handle,2,pxy);
        }

    for (dline = 0, xgrid = 39; dline < 2; dline++, xgrid += 160)
        {
        pxy[0] = 480;
        pxy[1] = xgrid;
        pxy[2] = 552;
        pxy[3] = xgrid;
        v_pline(handle,2,pxy);
        }

    for (dline = 0, xgrid = 39; dline < 2; dline++, xgrid += 160)
        {
        pxy[0] = 560;
        pxy[1] = xgrid;
        pxy[2] = 632;
        pxy[3] = xgrid;
        v_pline(handle,2,pxy);
        }


    }

/*
**  reserved for future use
*/


offmidi()
    {
    }
```

```
/*
**  This function is called at every measure beginning.
**  It was shortened in the ST MUSIC BOX program but
**  functioned just fine in the AUTO PLAYER.  We think that
**  is a real endorsement for 'C' and the 68000 chip that
**  all this can take place and not delay the player
**  by too much!!
*/


/*-----update the midi parameters--*/
/*-----must be el fasto code-----*/

extern      int port1, port2, port3, port4, vib1, vib2, vib3, vib4;
extern      int gprog1, gprog2, gprog3;

update()
    {
    print3d ( meascnt, 73, 8 );         /* measure count */
    print2d ( gprog, 76, 12 );          /* program selection */
    print2d ( gprog1, 76, 16 );         /* program selection */
    print2d ( gprog2, 76, 20 );         /* program selection */
    print2d ( gprog3, 76, 24 );         /* program selection */
    print2d ( gptime, 42, 6 );          /* portamento time */
    print2d ( timey, 50, 7 );           /* tempo -- demo here */

    if ( port1 != 0 )
        v_gtext ( handle, width*19, height*6,"ON ");
    else
        v_gtext ( handle, width*19, height*6,"OFF ");
    if ( vib1 != 0 )
        v_gtext ( handle, width*19, height*7,"ON ");
    else
        v_gtext ( handle, width*19, height*7,"OFF ");
    if ( vib2 != 0 )
        v_gtext ( handle, width*42, height*7,"ON ");
    else
        v_gtext ( handle, width*42, height*7,"OFF ");
    if ( vib3 != 0 )
        v_gtext ( handle, width*19, height*8,"ON ");
    else
        v_gtext ( handle, width*19, height*8,"OFF ");
    if ( vib4 != 0 )
        v_gtext ( handle, width*42, height*8,"ON ");
    else
        v_gtext ( handle, width*42, height*8,"OFF ");
    }
```

```
/*
**   turn up tempo
*/

/*
**   This function turns up the tempo with a key press.
**   This routine is called from the editor.
*/


upttt()
     {
     if ( timey < 99 )
          {
          timey++;
          tmpo_buffer[0] = tochar( timey );
          print2d( timey, 25, 21 );
          }
     }

/*
**   turn down tempo
*/

dnttt()
     {
     if ( timey > 3 )
          {
          timey--;
          tmpo_buffer[0] = tochar( timey );
          print2d( timey, 25, 21 );
          }
     }

/*
**   end of sets.c
*/
```

4.5 `btdata.c` source code

This file holds the data for the screen icons of the ST MUSIC BOX. The data is in two forms: 1) x & y coordinate information for a VDI '`v_pline`' call; and 2) the 'char' data which will be directly written to the screen.

The ST MUSIC BOX AUTO-PLAYER is designed to work in the medium resolution color mode. When bytes (called `chars` in 'C') are written to the screen, the way that they are written will determine their color.

In real life Len is far more than a black and white person, but when contributing his input to the design of the editor screen he screamed loud and clear that the screen should be black and white. Some at Xlent commented that to use only black and white on a beautiful color RGB monitor was a crime. No matter to Len, he felt that long hours at the monitor screen should be spend with as little eyestrain as possible. After jiggling the colors about we decided to leave the editor screen black and white, with the colors of the selected keys being highlighted in red.

Following the black and white theme we had to design the icon bit maps to appear in black and white. The reality of writing the code to have the icons appear black and white is easy, but the underlying reasons with roots in the bit plane structure of the color monitor isn't so easy. First we'll discuss the bit planes structure of the medium resolution mode and then move on to the byte arrangement.

The visible screen is made up of RAM (Random Access Memory). The BITS of the BYTES in the screen RAM determine what pixels (or picture elements) of the screen will be turned on. The BYTE arrangement is unique to each screen resolution mode. We will briefly discuss the BYTE arrangement of the monochrome and medium resolution modes.

In the high resolution monochrome mode, the screen resolution is 640 pixels wide by 400 pixels high. Each pixel width increment may be called a column and each pixel increment high may be called a row. Every row of 640 pixel columns is composed of 80 BYTES. Recalling the definition of BYTES from chapter 2, we know that each BYTE is made up of 8 BITS. Each of a BYTES 8 BITS can hold a 1 or a 0. It is simple to see that 80 BYTES times 8 BITS yields 640 pixels. If you surmised that every BIT in the monochrome mode equals a screen pixel you are correct!

```
MONOCHROME SCREEN BYTE ARRANGEMENT

|----------------------80 bytes-------------------|
|.....................640 pixels..................|
```

Monochrome mode is composed of an 80 BYTE row and there are 400 rows. The screen RAM required for the monochrome screen is 80 BYTES times 400 rows = 32000 BYTES.

The medium resolution mode screen BYTE arrangement is different from monochrome screen BYTE arrangement. In the medium resolution mode there are 4 colors available for each pixel. In the monochrome mode there were two colors available for each pixel. Four colors per pixel are permitted because the ST's operating system will assign TWO BITS for each pixel in the medium resolution mode.

MEDIUM RESOLUTION COLOR BIT OPTIONS

COLOR	BIT PATTERN
0	00
1	11
2	01
3	10

If you are familiar with the old Atari 130XE BIT arrangement it would be reasonable to assume that the two BITS required to generate the four color pixel would be located within the same BYTE. Unfortunately, that is not the case. The Atari ST uses a standard method of using the screen RAM to enable color choice called BIT PLANES.

The medium resolution color mode's screen pixel resolution is 640 by 200. Each row of BYTES contains 160 bytes. The total screen RAM required by the medium resolution color mode is 160 BYTES times 200 rows = 32000 BYTES. The same is required by the monochrome screen.

MEDIUM RESOLUTION BYTE/BIT ARRANGEMENT

```
Byte 1           Byte 2           Byte 3           Byte 4
BIT              BIT              BIT              BIT
7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0
```

SCREEN PIXEL LOCATION

X=0,y=0	uses	BIT	7	BYTE	1	&	BIT	7	BYTE 3
x=1,y=0	uses	BIT	6	BYTE	1	&	BIT	6	BYTE 3
x=2,y=0	uses	BIT	5	BYTE	1	&	BIT	5	BYTE 3
x=3,y=0	uses	BIT	4	BYTE	1	&	BIT	4	BYTE 3
x=4,y=0	uses	BIT	3	BYTE	1	&	BIT	3	BYTE 3
x=5,y=0	uses	BIT	2	BYTE	1	&	BIT	2	BYTE 3
x=6,y=0	uses	BIT	1	BYTE	1	&	BIT	1	BYTE 3
x=7,y=0	uses	BIT	0	BYTE	1	&	BIT	0	BYTE 3
x=8,y=0	uses	BIT	7	BYTE	2	&	BIT	7	BYTE 4
x=9,y=0	uses	BIT	6	BYTE	2	&	BIT	6	BYTE 4
x=10,y=0	uses	BIT	5	BYTE	2	&	BIT	5	BYTE 4
x=11,y=0	uses	BIT	4	BYTE	2	&	BIT	4	BYTE 4
x=12,y=0	uses	BIT	3	BYTE	2	&	BIT	3	BYTE 4
x=13,y=0	uses	BIT	2	BYTE	2	&	BIT	2	BYTE 4
x=14,y=0	uses	BIT	1	BYTE	2	&	BIT	1	BYTE 4
x=15,y=0	uses	BIT	0	BYTE	2	&	BIT	0	BYTE 4

You can see that the color for pixel location x=0 & y=0 is determined by the BIT 7 values from BYTES 1 & 3. The BITS ARE NOT CONTIGUOUS as per the arrangement in the Atari 130XE home computer. Note that the pixel locations all have the y=0 value. To write to the pixels directly below BYTE 1 you would need to add the value of 160 to BYTE 1's address. When you see the code in file 'btdata.c' you will see how easy it is to write BYTES below another in the 'C' programming language.

The 'C' source code for 'btdata.c'.

```
/*
**  btdata.c
*/

extern  long    orgscrn, screenl;
extern  int time, timex, timey, handle, dummy, height, width;

/*
**  boundary for changing note
**
**  this array of integers are to be used by a
**  VDI v_pline call.
*/

int pxynote[16] = {
        238+4,120,
        266+4,120,
        266+4,138,
        238+4,138,
        238+4,120,
        237+4,121,
        237+4,139,
        265+4,139 } ;

/*
**  boundary for T:00 info
**
**  this array of integers are to be used by a
**  VDI v_pline call.
*/

int pxyt00[16] = {
        179,160,
        179,170,
        215+4,170,
        215+4,160,
        179,160,
        178,161,
        178,171,
        214+4,171 } ;
```

```
/*
**  boundary for the tempo information
**
**  this array of integers are to be used by a
**  VDI v_pline call.
*/

int     pxytemp[16] = {
        56,180,
        56,194,
        214,194,
        214,180,
        56,180,
        55,181,
        55,195,
        213,195 } ;

/*
**  number 2 for time sig data
**
**  this array of integers is written directly into
**  screen memory
*/

char    sig2dat[13][4] = {
        255,0,255,0,
        34,0,34,0,
        65,0,65,0,
        1,0,1,0,
        1,0,1,0,
        2,0,2,0,
        255,0,255,0,
        8,0,8,0,
        16,0,16,0,
        32,0,32,0,
        92,0,92,0,
        227,0,227,0,
        255,0,255,0 } ;

/*
**  # 3 time sig data
**
**  this array of integers is written directly into
**  screen memory
*/
```

```
char     sig3dat[13][4] = {
         255,0,255,0,
         34,0,34,0,
         65,0,65,0,
         3,0,3,0,
         6,0,6,0,
         57,0,57,0,
         255,0,255,0,
         1,0,1,0,
         1,0,1,0,
         2,0,2,0,
         4,0,4,0,
         24,0,24,0,
         255,0,255,0 } ;

/*
**  #4 time sig data
**
**  this array of integers is written directly into
**  screen memory
*/

char     sig4dat[13][4] = {
         255,0,255,0,
         12,0,12,0,
         20,0,20,0,
         20,0,20,0,
         36,0,36,0,
         75,0,75,0,
         255,0,255,0,
         232,0,232,0,
         8,0,8,0,
         8,0,8,0,
         16,0,16,0,
         16,0,16,0,
         255,0,255,0 } ;

/*
**  #5 time sig data
**
**  this array of integers is written directly into
**  screen memory
*/

char     sig5dat[13][4] = {
         255,0,255,0,
         28,0,28,0,
        224,0,224,0,
        128,0,128,0,
        142,0,142,0,
        177,0,177,0,
        255,0,255,0,
        2,0,2,0,
        4,0,4,0,
        4,0,4,0,
        24,0,24,0,
        96,0,96,0,
        255,0,255,0 } ;

/*
**  turtle icon data
**
**  this array of integers is written directly into
**  screen memory
*/

char    turcon[11][4] = {
        0,0,0,0,
        0,0,0,0,
        14,0,14,0,
        57,128,57,128,
        115,192,115,192,
        103,204,103,204,
        255,252,255,252,
        255,224,255,224,
        32,64,32,64,
        48,96,48,96,
        0,0,0,0 };

/*
**  rabbit icon
**
**  this array of integers is written directly into
**  screen memory
*/

char    rabcon[11][4] = {
        0,0,0,0,
        12,0,12,0,
        8,0,8,0,
        16,0,16,0,
        35,224,35,224,
        117,84,117,84,
        126,168,126,168,
        37,88,37,88,
        3,240,3,240,
        2,16,2,16,
        4,32,4,32 };

/*
**  disk icon data
**
**  this array of integers is written directly into
**  screen memory
*/

char    dicon[20][8] = {
        199,255,199,255,255,252,255,252,
        204,0,204,0,0,4,0,4,
        204,0,204,0,0,4,0,4,
        204,255,204,255,255,132,255,132,
        204,240,204,240,3,196,3,196,
        204,240,204,240,115,228,115,228,
        204,240,204,240,115,228,115,228,
        204,240,204,240,115,228,115,228,
        204,240,204,240,3,228,3,228,
        204,255,204,255,255,228,255,228,
        204,255,204,255,255,228,255,228,
        204,255,204,255,255,228,255,228,
        204,255,204,255,255,228,255,228,
        204,255,204,255,255,228,255,228,
        204,255,204,255,255,228,255,228,
        204,255,204,255,255,228,255,228,
        204,0,204,0,0,4,0,4,
        204,0,204,0,0,4,0,4,
        207,255,207,255,255,252,255,252,
        207,255,207,255,255,248,255,248 } ;

/*
**  player icon
**
**  this array of integers is written directly into
**  screen memory
*/

char    playicon[20][12] = {
        127,255,127,255,255,255,255,255,255,255,255,255,
        192,0,192,0,0,0,0,0,0,1,0,1,
        192,0,192,0,0,0,0,0,0,1,0,1,
        192,0,192,0,0,7,0,7,249,1,249,1,
        192,192,192,192,1,124,1,124,9,1,9,1,
        192,192,192,192,30,0,30,0,4,129,4,129,
        192,192,192,192,224,0,224,0,4,129,4,129,
        192,195,192,195,0,0,0,0,2,65,2,65,
        192,236,192,236,14,130,14,130,42,65,42,65,
        192,232,192,232,14,135,14,135,58,65,58,65,
        192,236,192,236,8,239,8,239,146,65,146,65,
        192,195,192,195,0,0,0,0,2,65,2,65,
        192,192,192,192,224,0,224,0,4,129,4,129,
        192,192,192,192,30,0,30,0,4,129,4,129,
        192,192,192,192,1,124,1,124,9,1,9,1,
        192,0,192,0,0,7,0,7,249,1,249,1,
        192,0,192,0,0,0,0,0,1,0,1,
        192,0,192,0,0,0,0,0,1,0,1,
        255,255,255,255,255,255,255,255,255,255,255,255,
        255,255,255,255,255,255,255,255,255,254,255,254 } ;

/*
**  music icon data
**
**  this array of integers is written directly into
**  screen memory
*/

char    muicon[45][8] = {
        127,255,127,255,255,252,255,252,
        192,1,192,1,0,4,0,4,
        192,1,192,1,0,4,0,4,
        195,129,195,129,3,132,3,132,
        204,97,204,97,15,228,15,228,
        204,97,204,97,15,228,15,228,
        204,97,204,97,15,228,15,228,
        195,129,195,129,3,132,3,132,
        192,1,192,1,0,4,0,4,
        192,1,192,1,0,4,0,4,
        255,255,255,255,255,252,255,252,
        192,1,192,1,0,4,0,4,
        192,1,192,1,0,4,0,4,
        199,129,199,129,6,4,6,4,
        198,65,198,65,6,4,6,4,
        198,33,198,33,6,4,6,4,
        199,129,199,129,6,4,6,4,
        198,65,198,65,6,4,6,4,
        198,33,198,33,6,4,6,4,
        198,1,198,1,6,4,6,4,
        192,1,192,1,0,4,0,4,
        192,1,192,1,0,4,0,4,
        255,255,255,255,255,252,255,252,
        192,1,192,1,0,4,0,4,
        192,1,192,1,0,4,0,4,
        199,129,199,129,7,132,7,132,
        198,65,198,65,6,68,6,68,
        198,33,198,33,6,36,6,36,
        199,129,199,129,6,4,6,4,
        198,65,198,65,6,4,6,4,
        198,33,198,33,6,4,6,4,
        199,129,199,129,6,4,6,4,
        198,65,198,65,6,4,6,4,
        198,33,198,33,6,4,6,4,
        192,1,192,1,0,4,0,4,
        192,1,192,1,0,4,0,4,
        255,255,255,255,255,252,255,252,
        192,0,192,0,0,4,0,4,
        192,0,192,0,0,4,0,4,
        192,15,192,15,240,4,240,4,
        192,15,192,15,240,4,240,4,
        192,0,192,0,0,4,0,4,
        192,0,192,0,0,4,0,4,
        255,255,255,255,255,252,255,252,
        255,255,255,255,255,248,255,248 };

/*
**  outline the letter S - 8 points -
**
**  this array of integers are to be used by a
**  VDI v_pline call.
*/

int pxys[16] = {
        80,144,
        102,144,
        102,154,
        80,154,
        80,144,
        79,145,
        79,155,
        101,155 };

/*
**  outline the natural - 8 points -
**
**  this array of integers are to be used by a
**  VDI v_pline call.
*/
```

```
int pxynat[16] = {
        80,144+16,
        102,144+16,
        102,154+18,
        80,154+18,
        80,144+16,
        79,145+16,
        79,155+18,
        101,155+18 };

/*
**   draw the dot icon
**
**   this array of integers are to be used by a
**   VDI v_pline call.
*/

int ldotdraw[12] = {
        90+56,148+16,
        92+56,148+16,
        93+56,149+16,
        89+56,149+16,
        90+56,150+16,
        92+56,150+16 } ;

/*
**   drawnto natural symbol
**
**   this array of integers are to be used by a
**   VDI v_pline call.
*/

int natdraw[10] = {
        82,152+16,
        82,146+16,
        91,152+16,
        100,146+16,
        100,152+16 } ;

/*
**   outline the letter R - 8 points -
**
**   this array of integers are to be used by a
**   VDI v_pline call.
*/
```

```
int pxyr[16] = {
        80+56,144,
        102+56,144,
        102+56,154,
        80+56,154,
        80+56,144,
        79+56,145,
        79+56,155,
        101+56,155 };

/*
**  outline the dot - 8 points -
**
**  this array of integers are to be used by a
**  VDI v_pline call.
*/

int pxydot[16] = {
        80+56,144+16,
        102+56,144+16,
        102+56,154+16,
        80+56,154+16,
        80+56,144+16,
        79+56,145+16,
        79+56,155+16,
        101+56,155+16 };

/*
**  outline the letter B - 8 points -
**
**  this array of integers are to be used by a
**  VDI v_pline call.
*/

int pxyb[16] = {
        80+112,144,
        102+112,144,
        102+112,154,
        80+112,154,
        80+112,144,
        79+112,145,
        79+112,155,
        101+112,155 };
```

```
/*
**   outline for the letters U & B - 8 points -
**
**   this array of integers are to be used by a
**   VDI v_pline call.
*/

int pxyub[16] = {
        4,176,
        42,176,
        42,194,
        4,194,
        4,176,
        3,177,
        3,195,
        41,195 };

/*
**   outline for the large box - 8 points -
**
**   this array of integers are to be used by a
**   VDI v_pline call.
*/

int pxybb[16] = {
        1,118,
        286,118,
        286,198,
        1,198,
        1,118,
        0,119,
        0,199,
        285,199 };

/*
**   outline for M box - 17 points -
**
**   this array of integers are to be used by a
**   VDI v_pline call.
*/

int pxymt[34] = {
        67,120,
        114,120,
        114,131,
        106,131,
        106,132,
        113,132,
        106,132,
        106,138,
        73,138,
        73,131,
        67,131,
        67,120,
        66,121,
        66,132,
        72,132,
        72,139,
        105,139 };

/*
**   outline for V box - 17 points -
**
**   this array of integers are to be used by a
**   VDI v_pline call.
*/

int pxymv[34] = {
        67+56,120,
        114+56,120,
        114+56,131,
        106+56,131,
        106+56,132,
        113+56,132,
        106+56,132,
        106+56,138,
        73+56,138,
        73+56,131,
        67+56,131,
        67+56,120,
        66+56,121,
        66+56,132,
        72+56,132,
        72+56,139,
        105+56,139 };

/*
**   outline for O box - 17 points -
**
**   this array of integers are to be used by a
**   VDI v_pline call.
*/
```

```
int pxymo[34] = {
        67+112,120,
        114+112,120,
        114+112,131,
        106+112,131,
        106+112,132,
        113+112,132,
        106+112,132,
        106+112,138,
        73+112,138,
        73+112,131,
        67+112,131,
        67+112,120,
        66+112,121,
        66+112,132,
        72+112,132,
        72+112,139,
        105+112,139 };


/*
**   bass and cleff data
**
**   this array of integers is written directly into
**   screen memory
*/

char    ldcleff[65][4] = {
        0,96,0,96,
        0,224,0,224,
        0,144,0,144,
        1,152,1,152,
        1,152,1,152,
        3,24,3,24,
        3,24,3,24,
        3,24,3,24,
        255,255,255,255,
        3,48,3,48,
        1,48,1,48,
        0,224,0,224,
        0,192,0,192,
        1,128,1,128,
        255,255,255,255,
        7,128,7,128,
        14,128,14,128,
        28,128,28,128,
        24,64,24,64,
48,64,48,64,
255,255,255,255,
99,112,99,112,
102,88,102,88,
68,76,68,76,
68,44,68,44,
68,36,68,36,
255,255,255,255,
36,36,36,36,
34,44,34,44,
16,40,16,40,
8,48,8,48,
7,224,7,224,
255,255,255,255,
0,16,0,16,
2,16,2,16,
7,16,7,16,
15,48,15,48,
14,48,14,48,
7,224,7,224,
3,192,3,192,
0,0,0,0,
0,0,0,0,
0,0,0,0,
0,0,0,0,
255,255,255,255,
11,192,11,192,
16,224,16,224,
32,230,32,230,
40,118,40,118,
44,112,44,112,          /* either 4 or 44?? */
255,255,255,255,
28,112,28,112,
24,118,24,118,
0,230,0,230,
0,224,0,224,
0,192,0,192,
255,255,255,255,
0,192,0,192,
1,128,1,128,
3,0,3,0,
6,0,6,0,
8,0,8,0,
255,255,255,255,
32,0,32,0,
64,0,64,0 } ;
```

```
/*
**   define 1 for mono and 2 for med rez col
*/

#define     blgdval     2

/*
**   data for piao keyboard shape
**
**   this array of integers are to be used by a
**   VDI v_pline call.
*/

int bkdat [100] = {
         435, ( 380 + 16 ) / blgdval,
         435, ( 220 + 16 ) / blgdval,
         405, ( 220 + 16 ) / blgdval,
         405, ( 300 + 16 ) / blgdval,
         390, ( 300 + 16 ) / blgdval,
         390, ( 380 + 16 ) / blgdval,
         390, ( 300 + 16 ) / blgdval,
         375, ( 300 + 16 ) / blgdval,
         375, ( 220 + 16 ) / blgdval,
         360, ( 220 + 16 ) / blgdval,
         360, ( 300 + 16 ) / blgdval,
         345, ( 300 + 16 ) / blgdval,
         345, ( 380 + 16 ) / blgdval,
         345, ( 300 + 16 ) / blgdval,
         330, ( 300 + 16 ) / blgdval,
         330, ( 220 + 16 ) / blgdval,
         300, ( 220 + 16 ) / blgdval,
         300, ( 380 + 16 ) / blgdval,
         480, ( 380 + 16 ) / blgdval,
         480, ( 300 + 16 ) / blgdval,
         465, ( 300 + 16 ) / blgdval,
         465, ( 220 + 16 ) / blgdval,
         495, ( 220 + 16 ) / blgdval,
         495, ( 300 + 16 ) / blgdval,
         480, ( 300 + 16 ) / blgdval,
         480, ( 380 + 16 ) / blgdval,
         525, ( 380 + 16 ) / blgdval,
         525, ( 300 + 16 ) / blgdval,
         510, ( 300 + 16 ) / blgdval,
         510, ( 220 + 16 ) / blgdval,
         540, ( 220 + 16 ) / blgdval,
         540, ( 300 + 16 ) / blgdval,
         525, ( 300 + 16 ) / blgdval,
        525, ( 380 + 16 ) / blgdval,
        570, ( 380 + 16 ) / blgdval,
        570, ( 300 + 16 ) / blgdval,
        555, ( 300 + 16 ) / blgdval,
        555, ( 220 + 16 ) / blgdval,
        585, ( 220 + 16 ) / blgdval,
        585, ( 300 + 16 ) / blgdval,
        570, ( 300 + 16 ) / blgdval,
        570, ( 380 + 16 ) / blgdval,
        615, ( 380 + 16 ) / blgdval,
        615, ( 220 + 16 ) / blgdval,
        330, ( 220 + 16 ) / blgdval
         };

/*
**  draw the piano keyboard for editor
**
*/

bones()
    {
    int lddx, ldpxy4[4];
    v_pline( handle, 45, bkdat );
    v_pline( handle,8,pxyub );
    v_pline( handle,8,pxybb );
    v_pline( handle,8,pxys );
    v_pline( handle,8,pxyr );
    v_pline( handle,8,pxyb );
    v_pline( handle,8,pxynote );
    v_pline( handle,17,pxymt );
    v_pline( handle,17,pxymv );
    v_pline( handle,17,pxymo );
    ldtrebf();              /* print bass and treble */
    ldmuic();               /* print the music icons */
    lddicon();              /* print the disk icon */
    ldturab();              /* print turtle and rabbit */
    ldplicon();             /* print player icon */
    v_gtext( handle, width*23, height*21, "T:   ");
    v_pline( handle, 8, pxyt00 );   /* box around T:00 */
    ldstemp();              /* print tempo # */
    v_pline( handle, 8, pxytemp );
    v_pline( handle, 5, natdraw );
    v_pline( handle, 8, pxynat );
    v_pline( handle, 8, pxydot );
    v_pline( handle, 6, ldotdraw );
    ldpxy4[1] = 180;
    ldpxy4[3] = 194;
    for ( lddx = 82; lddx <= 184; lddx += 2 )
        {
        ldpxy4[0] = lddx;
        ldpxy4[2] = lddx;
        v_pline( handle, 2, ldpxy4 );
        }
    switch ( time )
        {
        case 16:
            ldti24();   /* time is 2/4 */
            break;
        case 24:
            ldti34();
            break;
        case 32:
            ldti44();
            break;
        case 40:
            ldti54();
            break;
        }
    graf_mouse( 3, &dummy );
    v_show_c( handle, 1 );
    }

/*
**  show the time signature for 2/4
**
**  This routine write byte values directly into screen memory
*/

ldti24()
    {
    int ldrow, ldcol, ldl;
    char    *ldptr, ldc1, ldc2;
    ldptr = screen1 + ( 39 * 160 ) + 16;
    /* 39 = row */
    for ( ldl = 0, ldrow = 0; ldrow < 13; ldl += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig2dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ldl + ldcol ) | ldc1;
            *( ldptr + ldl + ldcol ) = ldc2;
            }
    ldptr = screen1 + ( 51 * 160 ) + 16;
    for ( ldl = 0, ldrow = 0; ldrow < 13; ldl += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
```

```
            {
            ldc1 = sig4dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    ldptr = screen1 + ( 75 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig2dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    ldptr = screen1 + ( 87 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig4dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    }


/*
**  show the time signature for 3/4
*/

ldti34()
    {
    int ldrow, ldcol, ld1;
    char    *ldptr, ldc1, ldc2;
    ldptr = screen1 + ( 39 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig3dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    ldptr = screen1 + ( 51 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig4dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    ldptr = screen1 + ( 75 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig3dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    ldptr = screen1 + ( 87 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig4dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    }


/*
**  show the time signature for 4/4
*/

ldti44()
    {
    int ldrow, ldcol, ld1;
    char    *ldptr, ldc1, ldc2;
    ldptr = screen1 + ( 39 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig4dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    ldptr = screen1 + ( 51 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig4dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    ldptr = screen1 + ( 75 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
```

```
                {
                ldc1 = sig4dat[ldrow][ldcol];
                ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
                *( ldptr + ld1 + ldcol ) = ldc2;
                }
        ldptr = screen1 + ( 87 * 160 ) + 16;
        for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
            for ( ldcol = 0; ldcol < 4; ldcol++ )
                {
                ldc1 = sig4dat[ldrow][ldcol];
                ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
                *( ldptr + ld1 + ldcol ) = ldc2;
                }
        }


/*
**  show the time signature for 5/4
*/

ldti54()
        {
        int ldrow, ldcol, ld1;
        char    *ldptr, ldc1, ldc2;
        ldptr = screen1 + ( 39 * 160 ) + 16;
        for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
            for ( ldcol = 0; ldcol < 4; ldcol++ )
                {
                ldc1 = sig5dat[ldrow][ldcol];
                ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
                *( ldptr + ld1 + ldcol ) = ldc2;
                }
        ldptr = screen1 + ( 51 * 160 ) + 16;
        for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
            for ( ldcol = 0; ldcol < 4; ldcol++ )
                {
                ldc1 = sig4dat[ldrow][ldcol];
                ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
                *( ldptr + ld1 + ldcol ) = ldc2;
                }
        ldptr = screen1 + ( 75 * 160 ) + 16;
        for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
            for ( ldcol = 0; ldcol < 4; ldcol++ )
                {
                ldc1 = sig5dat[ldrow][ldcol];
                ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
                *( ldptr + ld1 + ldcol ) = ldc2;
```

```
          }
    ldptr = screen1 + ( 87 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 13; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = sig4dat[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    }


/*
**  show the tempo held in timey
*/

ldstemp()
    {
    print2d( timey, 25, 21 );
    }

/*
**  draw the treble and bass cleffs
*/

ldtrebf()
    {
    int ldrow, ldcol, ld1;
    char    *ldptr, ldc1, ldc2;
    ldptr = screen1 + ( 31 * 160 ) + 8;
    for ( ld1 = 0, ldrow = 0; ldrow < 65; ld1 += 160, ldrow++ )
        {
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = ldcleff[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
        }
    switch ( time )
        {
        case 16:
            ldti24();    /* time is 2/4 */
            break;
        case 24:
            ldti34();
            break;
        case 32:
            ldti44();
            break;
        case 40:
            ldti54();
            break;
        }
    }

/*
**  draw the music icons
*/

ldmuic()
    {
    int ldrow, ldcol, ld1;
    char    *ldptr, ldc1, ldc2;
    ldptr = screen1 + ( 144 * 160 ) + 60;
    for ( ld1 = 0, ldrow = 0; ldrow < 45; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 8; ldcol++ )
            {
            ldc1 = muicon[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    }


/*
**  draw the disk icon
*/

lddicon()
    {
    int ldrow, ldcol, ld1;
    char    *ldptr, ldc1, ldc2;
    ldptr = screen1 + ( 120 * 160 );
    for ( ld1 = 0, ldrow = 0; ldrow < 20; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 8; ldcol++ )
            {
            ldc1 = dicon[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    }
```

```
/*
**   draw the player icon
*/

ldplicon()
    {
    int ldrow, ldcol, ld1;
    char    *ldptr, ldc1, ldc2;
    ldptr = screen1 + ( 148 * 160 ) + 4;
    for ( ld1 = 0, ldrow = 0; ldrow < 20; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 12; ldcol++ )
            {
            ldc1 = playicon[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    }

/*
**   draw the rabbit and turtle icons
*/

ldturab()
    {
    int ldrow, ldcol, ld1;
    char    *ldptr, ldc1, ldc2;
    ldptr = screen1 + ( 181 * 160 ) + 16;
    for ( ld1 = 0, ldrow = 0; ldrow < 11; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = turcon[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    ldptr = screen1 + ( 181 * 160 ) + 48;
    for ( ld1 = 0, ldrow = 0; ldrow < 11; ld1 += 160, ldrow++ )
        for ( ldcol = 0; ldcol < 4; ldcol++ )
            {
            ldc1 = rabcon[ldrow][ldcol];
            ldc2 =  *( ldptr + ld1 + ldcol ) | ldc1;
            *( ldptr + ld1 + ldcol ) = ldc2;
            }
    }

/*
**   end of btdata.c
*/
```

4.6 `interf.c` source code

This file contains the information for the file structure of the AUTO PLAYER and also the ST MUSIC BOX's editor. We offer this information to the public so any parties interested in writing utility programs which will link differing MIDI programs will be able to do so. It is our belief that open architecture programs benefit mass segments of the computer community.

The ST MUSIC BOX's editor needs to hold much more information than the AUTO-PLAYER does. The AUTO-PLAYER's operation is time critical so we decided to handle all of the file conversion procedures before the player takes over. That is the reason for the slight wait after the editor's data loads into the machine.

```
/**interf.c*/

/*
**  definitions iincluded here
*/

#include    <a:portab.h>
#include    <a:machine.h>
#include    <a:obdefs.h>
#include    <a:define.h>
#include    <a:gemdefs.h>
#include    <a:osbind.h>

/*  externals here      */

extern  int wbox, hbox, dur1, dur2, dur3, port_state;
extern  int dur4,dur5,dur6,dur7,dur8;
extern  int contrl[12], intin[256], ptsin[256], ptsout[256];
extern  int intin[256], intout[256],pxy[10];
extern  int newbyte[8], oldbyte[8], typedone, hbox1, wbox1, hchar1;
extern  int wchar1;
extern  int l_intin[20], l_out[128], l_ptsin[20];
extern  int ev_kreturn,i,iter,gr_mkmx,gr_mkmy,groption,stylem,endm;
extern  int mode,set_mode;
extern  int set_font,handle,dummy,gr_mkmstate,gr_mkkstate,prtmem;
extern  int holdrez,set_effect;
extern  int holdx,holdy,whesc,whesc1,color_index,colorm,holdclr;
extern  int gr_mkresvd,intstyle,topbot,height,width;
extern  int exflag,widd,mull,radius;
extern  int begang,endang,yradius,x_boundary,y_boundary,k,k1,k2,k3,k4;
extern  int n,u,ab,kk,row,count,column,sp,mod,byte,j,quot,rem,demo;
extern  int note1, note2, note3, note4, note5, note6, note7, note8;
extern  int scrold, scrnew, mult, ymult, findex;
extern  int fcolor, linev, lineh, linesw;
extern  int byte1, byte2, ig, sgflag,prtsw,path;
extern  int timex, timey, first, octup;
extern  int octsw0, octsw1, octsw2, octsw3, octsw4, octsw5, octsw6;
extern  int octsw7, meascnt, cnt32;
extern  int chancnt, progcnt;
extern  int gfreq1, gfreq2, gfreq3, gfreq4,gfreq5,gfreq6,gfreq7,gfreq8;
extern  int gpsw, gptime, dblo1, gvibsw, gprog;
extern  int st_m;
extern  long    *ptr1, *ptr2, *ptr3, *ptr4, *ptr5, *ptr6, *ptr7, *ptr8;
extern  char    c;
extern  long    logicbase,screen1,screen2,orgscrn,ltemp;
extern  char    buffer1[32767];     /* main display screen */
extern  char    nbuff1[1024],nbuff2[1024],nbuff3[1024],nbuff4[1024];
extern  char    nbuff5[1024],nbuff6[1024],nbuff7[1024],nbuff8[1024];
extern  char    nbuff9[1024],nbuff10[1024],nbuff11[1024],nbuff12[1024];
extern  char    nbuff13[1024],nbuff14[1024],nbuff15[1024],nbuff16[1024];


/*
**  These buffers will hold the music data for the
**  AUTO-PLAYER.  The dtab buffers hold note duration
**  data.  The ftab buffers hold the frequency values
**  and the vtab buffers are reserved for future use.
*/


extern  int dtab1[1024], dtab2[1024],dtab3[1024],
        dtab4[1024], dtab5[1024],dtab6[1024],
        dtab7[1024], dtab8[1024];

extern  int ftab1[1024], ftab2[1024],ftab3[1024],
        ftab4[1024], ftab5[1024],ftab6[1024],
        ftab7[1024], ftab8[1024];

extern  int vtab1[1], vtab2[1],vtab3[1],
        vtab4[1], vtab5[1],vtab6[1],
        vtab7[1], vtab8[1];

extern  int ldtmpo[ 264 ];
extern  int progtab,chan1_buffer[264],chan2_buffer[264];
extern  int chan3_buffer[264], chan4_buffer[264];
extern  int port1, port2, port3, port4, vib1, vib2, vib3, vib4;
extern  int xkey;
extern  char    tmpo_buffer[ 264 ];


/*
**  The prog0 buffers hold the INSTRUMENT VOICE number.  These
**  values are passed to the MIDI synthesizer at the beginning
**  of each measure.
*/


int prog0c[264], prog1c[264], prog2c[264], prog3c[264];

int fflag1, fflag2, fflag3, fflag4, fflag5, fflag6, fflag7, fflag8;

/*
*   reserved for future use
*
*/

showb2()
    {
    }

/*
**  adjust all out of bounds freq vals to rests
**
**  This routine is sort of a broad spectrum anti-biotic
**  routine.  There shoud NEVER be any out of bounds values,
**  but we figured if any ever appeared ('the best laid plans...etc')
**  this routine would clean them up!
*/


ldadjrest()
    {
    int     ldcnt;
    for ( ldcnt = 0; ldcnt < 1024; ldcnt++ )
        {
        if ( ftab1[ ldcnt ] > 96 )
            ftab1[ ldcnt ] = 1;
        if ( ftab2[ ldcnt ] > 96 )
            ftab2[ ldcnt ] = 1;
        if ( ftab3[ ldcnt ] > 96 )
            ftab3[ ldcnt ] = 1;
        if ( ftab4[ ldcnt ] > 96 )
            ftab4[ ldcnt ] = 1;
        if ( ftab5[ ldcnt ] > 96 )
            ftab5[ ldcnt ] = 1;
        if ( ftab6[ ldcnt ] > 96 )
            ftab6[ ldcnt ] = 1;
        if ( ftab7[ ldcnt ] > 96 )
            ftab7[ ldcnt ] = 1;
        if ( ftab8[ ldcnt ] > 96 )
            ftab8[ ldcnt ] = 1;
        }
    }

/*
*   reserved for future use
*/
ttime1()
    {
    timex = 180;
    timey = 60;
    }

/*
**  clear the ftab and dtab buffers
*/ .

cfdtab()
    {
    int num4;
    for ( num4 = 0; num4 < 1024; num4++ )
        {
        ftab1[ num4 ] = 0;
        ftab2[ num4 ] = 0;
        ftab3[ num4 ] = 0;
        ftab4[ num4 ] = 0;
        ftab5[ num4 ] = 0;
        ftab6[ num4 ] = 0;
        ftab7[ num4 ] = 0;
        ftab8[ num4 ] = 0;
        dtab1[ num4 ] = 0;
        dtab2[ num4 ] = 0;
        dtab3[ num4 ] = 0;
        dtab4[ num4 ] = 0;
        dtab5[ num4 ] = 0;
```

```
            dtab6[ num4 ] = 0;
            dtab7[ num4 ] = 0;
            dtab8[ num4 ] = 0;
            }
        }

/*
**   adjust duration for tie
*/

adjtie()
        {
        int tctr;
        for ( tctr = 0; tctr < 1024; tctr++ )
            {
            if ( dtab1[tctr] == 0x65 || dtab1[tctr] == 0x76 )
                dtab1[tctr] = 0;
            if ( dtab2[tctr] == 0x65 || dtab2[tctr] == 0x76 )
                dtab2[tctr] = 0;
            if ( dtab3[tctr] == 0x65 || dtab3[tctr] == 0x76 )
                dtab3[tctr] = 0;
            if ( dtab4[tctr] == 0x65 || dtab4[tctr] == 0x76 )
                dtab4[tctr] = 0;
            if ( dtab5[tctr] == 0x65 || dtab5[tctr] == 0x76 )
                dtab5[tctr] = 0;
            if ( dtab6[tctr] == 0x65 || dtab6[tctr] == 0x76 )
                dtab6[tctr] = 0;
            if ( dtab7[tctr] == 0x65 || dtab7[tctr] == 0x76 )
                dtab7[tctr] = 0;
            if ( dtab7[tctr] == 0x65 || dtab8[tctr] == 0x76 )
                dtab8[tctr] = 0;
            }
        }
/*
*   sequencer to be called from the editor
*/

playit()
        {
        int ldt1, ldt2;
        v_hide_c( handle );
        holdrez=Getrez();              /*get monitor rez*/

        logicbase=Logbase();           /*get logical base*/
        orgscrn=Physbase();            /*get scratch screen addr*/
        screen1=(0xffff00&buffer1);    /*top screen addr*/
```

```
screen1+=(0x100);          /*adj ptr*/
ptr1 = screen1;
ptr2 = screen1;
ptr3 = screen1;
ptr4 = screen1;
ptr5 = screen1;
ptr6 = screen1;
ptr7 = screen1;
ptr8 = screen1;
vsf_perimeter(handle,1);       /*edge of shape visible*/
Setscreen ( screen1, screen1, -1 );
v_clrwk(handle);               /* clear screen */
prtdemo();                     /* print demo grids & text */
cfdtab();                      /* clr dtab and ftab buffers */
conf1();                       /* rem-- 1 thru 8 here!! */
conf2();
conf3();
conf4();
conf5();
conf6();
conf7();
conf8();
clr03();                       /* write 0s to prog#c buffs */
conc0();                       /* chan#_buffer -> prog#c */
conc1();
conc2();
conc3();
ldtrans();         /* trasnspose freq data */
ldadjrest();
adjtie();          /* adjust dur buffers for tie */
ldgetst();         /* adjust duration buffers */
exflag = 0;
while ( exflag == 0 )
    {
    for ( mod = 0; mod < 1; mod++ ) /* play once */
        {
        first = 0;
        gpsw = 1;              /*for demo*/
        cnt32 = 0;
        progcnt = 0;
        meascnt = st_m;
        chport ( 1, port1, 0 ); /* port for chan # 0 */
        chport ( 1, port2, 1 ); /* port for chan # 1 */
        chport ( 1, port3, 2 ); /* port for chan # 2 */
        chport ( 1, port4, 3 ); /* port for chan # 3 */
        chvib  ( vib1, 0 ); /* cc 1 */
        chvib  ( vib2, 1 ); /* cc 1 */
        chvib  ( vib3, 2 ); /* cc 1 */
        chvib  ( vib4, 3 ); /* cc 1 */
        chprog ( 0, prog0c[0] ); /* init program change #1 */
        chprog ( 1, prog1c[0] );
        chprog ( 2, prog2c[0] );
        chprog ( 3, prog3c[0] );
        port_state = Giaccess ( port_state, 0x07 );
        n = port_state & 0xf8;  /*turn on 3 voices*/
        n |= (0x38);            /*no noise here*/
        Giaccess ( n, 0x87 );   /*set GI sound chip */

        if ( mod == 0 )      /* set asdr */
            setenv ( 20000, 20000, 0 );
        else
            setenv ( 2, 2, 6 );
        dur1 = 0;        /* set to default */
        dur2 = 0;
        dur3 = 0;
        dur4 = 0;
        dur5 = 0;
        dur6 = 0;
        dur7 = 0;
        dur8 = 0;
        note1 = 0;
        note2 = 0;
        note3 = 0;
        note4 = 0;
        note5 = 0;
        note6 = 0;
        note7 = 0;
        note8 = 0;
        playmus();                    /* prelude in c minor */
        Giaccess( 0x00, 0x88 );       /* off snd */

        Giaccess( 0x00, 0x89 );       /* off snd */

        Giaccess( 0x00, 0x8a );       /* off snd */

        Giaccess ( port_state, 0x87 ); /* restore port */

        }
    v_clrwk( handle );
    Setscreen ( orgscrn, orgscrn, -1 );
    }
```

```
/*
**  turn off all the notes
**
*/

    for ( ldt2 = 0; ldt2 < 4; ldt2++ )
        {
        for ( ldt1 = 36; ldt1 < 97; ldt1++ )
            {
            Bconout ( 3, 128 + ldt2 ); /* chan */
            Bconout ( 3, ldt1 );     /* freq */
            Bconout ( 3, 0 );    /* vel = 0 = off */
            }
        }
    }

/*
**  zero out prog#c buffers
*/

clr03()
    {
    int cnt03;
    for ( cnt03 = 0; cnt03 < 264; cnt03++ )
        {
        prog0c[ cnt03 ]  = 0;
        prog1c[ cnt03 ]  = 0;
        prog2c[ cnt03 ]  = 0;
        prog3c[ cnt03 ]  = 0;
        }
    }

/*
**  convert chan1_buffer to prog0c buffer
*/

conc0()
    {
    int cnt03, t03;
    if ( chan1_buffer[0] == 0 ) /* auto default check */
        chan1_buffer[0] = 1;
    for ( cnt03 = 0; cnt03 < 264; cnt03++ )
        {
        t03 = chan1_buffer[ cnt03 ];
        if ( t03 != 0 )
            prog0c[ cnt03 ] = t03 - 1;
        else
            {
            cnt03--;
            t03 = prog0c[ cnt03 ];
            cnt03++;
            prog0c[ cnt03 ] = t03;
            }
        }
    }


/*
**  convert chan2_buffer to prog1c buffer
*/

conc1()
    {
    int cnt03, t03;
    if ( chan2_buffer[0] == 0 ) /* auto default check */
        chan2_buffer[0] = 1;
    for ( cnt03 = 0; cnt03 < 264; cnt03++ )
        {
        t03 = chan2_buffer[ cnt03 ];
        if ( t03 != 0 )
            prog1c[ cnt03 ] = t03 - 1;
        else
            {
            cnt03--;
            t03 = prog1c[ cnt03 ];
            cnt03++;
            prog1c[ cnt03 ] = t03;
            }
        }
    }


/*
**  convert chan3_buffer to prog2c buffer
*/

conc2()
    {
    int cnt03, t03;
    if ( chan3_buffer[0] == 0 ) /* auto default check */
        chan3_buffer[0] = 1;
    for ( cnt03 = 0; cnt03 < 264; cnt03++ )
```

```
        {
        t03 = chan3_buffer[ cnt03 ];
        if ( t03 != 0 )
            prog2c[ cnt03 ] = t03 - 1;
        else
            {
            cnt03--;
            t03 = prog2c[ cnt03 ];
            cnt03++;
            prog2c[ cnt03 ] = t03;
            }
        }
    }


/*
**  convert chan4_buffer to prog3c buffer
*/

conc3()
    {
    int cnt03, t03;
    if ( chan4_buffer[0] == 0 ) /* auto default check */
        chan4_buffer[0] = 1;
    for ( cnt03 = 0; cnt03 < 264; cnt03++ )
        {
        t03 = chan4_buffer[ cnt03 ];
        if ( t03 != 0 )
            prog3c[ cnt03 ] = t03 - 1;
        else
            {
            cnt03--;
            t03 = prog3c[ cnt03 ];
            cnt03++;
            prog3c[ cnt03 ] = t03;
            }
        }
    }


/*
**  convert tmpo_buffer to ldtmpo buffer
*/

ldgetst()
    {
```

```
    int cnt03, t03;
    int ldxxx[ 264 ];
    char    hc1;
    for ( cnt03 = 0; cnt03 < 264; cnt03++ )
        {
        hc1 = tmpo_buffer[ cnt03 ];
        t03 = toint( hc1 );
        ldxxx[ cnt03 ] = t03;
        }

        if ( ldxxx[0] == 0 )
            ldxxx[0] = 99;

    for ( cnt03 = 0; cnt03 < 264; cnt03++ )
        {
        t03 = ldxxx[ cnt03 ];
        if ( t03 != 0 )
            ldtmpo[ cnt03 ] = t03;
        else
            {
            cnt03--;
            t03 = ldtmpo[ cnt03 ];
            cnt03++;
            ldtmpo[ cnt03 ] = t03;
            }
        }
    }

/*
**  signed chars need to be converted to integers in alcyon 'C'
**  and therefor this routine - groan!
*/


/*
*   character to integer conversion
*/

toint ( c3 )
    char    c3;
    {
    char    c4;
    int il1;
    il1 = 0;
    c4 = 0;
    while ( c3 != c4 )
```

```
        {
        c4++;
        ill++;
        }
    return ( ill );
    }

/*
*   integer to character conversion
*/

tochar ( c3i )
    int     c3i;
    {
    int c4;
    char    ill;
    ill = 0;
    c4 = 0;
    while ( c3i != c4 )
        {
        c4++;
        ill++;
        }
    return ( ill );
    }

/*
**  Here is a key file conversion routine.  The editor  uses the
**  values of 49, 55, 66, and 72 to stand for a whole note.
**  The value returned goes into the duration buffer.
**  NOTE: in futute versions of the editor we are planning
**  to change the whole note value to 96 (instead of the 32).
**  All other duration values will also be multiplied by
**  three allowing the use of quarter note and eighth note
**  triplets.
*/


/*
 *  integer to duration value
 */

dequ( il7 )
    int il7;
    {
    switch ( il7 )
    {
    case 49:
    case 55:
    case 66:
    case 72:
        il7 = 32;
        break;
    case 50:
    case 56:
    case 67:
    case 73:
        il7 = 16;
        break;
    case 51:
    case 57:
    case 68:
    case 74:
        il7 = 8;
        break;
    case 52:
    case 58:
    case 69:
    case 75:
        il7 = 4;
        break;
    case 53:
    case 59:
    case 70:
    case 76:
        il7 = 2;
        break;
    case 54:
    case 60:
    case 71:
    case 77:
        il7 = 1;
        break;
    case 100:
    case 117:
        il7 = 100;
        break;
    case 0x65:       /* tie fix */
    case 0x76:
        il7 = 0x65;
        break;
    default:
            il7 = 255;        /* bypass sharp and flat */
        }
    return ( il7 );
    }

/*
*   call to convert editor buffs to
*   sequencer buffs
*/

conf1()
    {
    char    cc1;
    cc1 = nbuff2[ 0 ];
    switch ( cc1 )
        {
        case 0x00:
            fflag1 = 0;
            break;
        default:
            fflag1 = 1;
            cvf1();
            cvd1();
            break;
        }
    }

/*
**  transpose frequency data
*/

ldtrans()
    {
    int trdum, ctrt, ttt;
    trdum = xkey - 25;  /* set value */
    for ( ctrt = 0; ctrt < 1024; ctrt++ )
        {
        if ( ftab1[ ctrt ] >= 2 )
                {
                ttt = ftab1[ ctrt ] - trdum;
                if ( ttt < 2 || ttt > 96 )
                    ttt = 1;    /* forced rest */
                ftab1[ ctrt ] = ttt;
                }
        if ( ftab2[ ctrt ] >= 2 )
                {
```

```
                ttt = ftab2[ ctrt ] - trdum;
                if ( ttt < 2 || ttt > 96 )
                    ttt = 1;     /* forced rest */
                ftab2[ ctrt ] = ttt;
                }
        if ( ftab3[ ctrt ] >= 2 )
                {
                ttt = ftab3[ ctrt ] - trdum;
                if ( ttt < 2 || ttt > 96 )
                    ttt = 1;     /* forced rest */
                ftab3[ ctrt ] = ttt;
                }
        if ( ftab4[ ctrt ] >= 2 )
                {
                ttt = ftab4[ ctrt ] - trdum;
                if ( ttt < 2 || ttt > 96 )
                    ttt = 1;     /* forced rest */
                ftab4[ ctrt ] = ttt;
                }
        if ( ftab5[ ctrt ] >= 2 )
                {
                ttt = ftab5[ ctrt ] - trdum;
                if ( ttt < 2 || ttt > 96 )
                    ttt = 1;     /* forced rest */
                ftab5[ ctrt ] = ttt;
                }
        if ( ftab6[ ctrt ] >= 2 )
                {
                ttt = ftab6[ ctrt ] - trdum;
                if ( ttt < 2 || ttt > 96 )
                    ttt = 1;     /* forced rest */
                ftab6[ ctrt ] = ttt;
                }
        if ( ftab7[ ctrt ] >= 2 )
                {
                ttt = ftab7[ ctrt ] - trdum;
                if ( ttt < 2 || ttt > 96 )
                    ttt = 1;     /* forced rest */
                ftab7[ ctrt ] = ttt;
                }
        if ( ftab8[ ctrt ] >= 2 )
                {
                ttt = ftab8[ ctrt ] - trdum;
                if ( ttt < 2 || ttt > 96 )
                    ttt = 1;     /* forced rest */
                ftab8[ ctrt ] = ttt;
```

```
                    }
                }
            }

/*
*   convert measure freq data
*   to sequencer freq data
*/

cvf1()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1;
    xflag1 = 0;
    ctr = 0;     /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff2[ mctr ];
        switch ( c1 )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff2[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    test = toint ( bitter );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        ftab1[ ctr ] = test;
                        nctr++;
```

```
                            ctr++;
                            }
                        break;
                    }
                    }
            }
            mctr += 24;
            }
        }

/*
*   convert measure freq data
*   to sequencer duration data
*/

cvd1()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1, tempd1;
    xflag1 = 0;
    ctr = 0;    /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff1[ mctr ];
        switch ( c1 )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff1[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    tempd1 = toint ( bitter );
                    test = dequ ( tempd1 );
                    if ( test >= 128 )
```

```
                        {
                        nctr++;
                        }
                        else
                        {
                        switch ( test )
                        {
                        case    100:
                        case    117:
                            ctr--;  /* previous */
                            il1=dtabl[ctr]; /*dur*/
                            ctr++;
                            dtabl[ctr] = il1/2;
                            ctr++; /*restore dur*/
                            nctr++; /*new nbuff*/
                            break;
                        default:
                            dtabl[ ctr ] = test;
                            nctr++;
                            ctr++;
                            break;
                        }
                        }
                    break;
                }
                }
        }
        mctr += 24;
        }
    }

/*
*   call to convert editor buffs to
*   sequencer buffs
*/

conf2()
    {
    char    cc1;
    cc1 = nbuff4[ 0 ];
    switch ( cc1 )
        {
        case 0x00:
            fflag1 = 0;
            break;
        default:
            fflag1 = 1;
            cvf2();
            cvd2();
            break;
        }
    }

/*
*   convert measure freq data
*   to sequencer freq data
*/

cvf2()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1;
    xflag1 = 0;
    ctr = 0;    /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff4[ mctr ];
        switch ( c1 )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff4[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    test = toint ( bitter );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        ftab2[ ctr ] = test;
                        nctr++;
                        ctr++;
                        }
                    break;
                }
                }
        }
        mctr += 24;
        }
    }

/*
*   convert measure freq data
*   to sequencer duration data
*/

cvd2()
    {
    char    bitter, cl;
    int mctr, ctr, nctr, xflag1, xflag2, test, ill, tempd1;
    xflag1 = 0;
    ctr = 0;    /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        cl = nbuff3[ mctr ];
        switch ( cl )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff3[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    tempdl = toint ( bitter );
                    test = dequ ( tempdl );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        switch ( test )
                        {
                        case    100:
                        case    117:
                            ctr--;  /* previous */
                            il1=dtab2[ctr]; /*dur*/
                            ctr++;
                            dtab2[ctr] = il1/2;
                            ctr++; /*restore dur*/
                            nctr++; /*new nbuff*/
                            break;
                        default:
                            dtab2[ ctr ] = test;
                            nctr++;
                            ctr++;
                            break;
                        }
                        }
                    break;
                }
                }
        }
        mctr += 24;
        }
    }

/*
*   call to convert editor buffs to
*   sequencer buffs
*/

conf3()
    {
    char    cc1;
    cc1 = nbuff6[ 0 ];
    switch ( cc1 )
        {
        case 0x00:
                fflag1 = 0;
                break;
            default:
                fflag1 = 1;
                cvf3();
                cvd3();
                break;
            }
        }

/*
*   convert measure freq data
*   to sequencer freq data
*/

cvf3()
        {
        char    bitter, c1;
        int mctr, ctr, nctr, xflag1, xflag2, test, il1;
        xflag1 = 0;
        ctr = 0;     /* ftab element # */
        mctr = 0;
        mctr = ( st_m - 1 ) * 24;
        while ( xflag1 == 0 )
            {
            c1 = nbuff6[ mctr ];
            switch ( c1 )
            {
            case 0x00:
                xflag1 = 1;
                break;
            default:
                nctr = 0;    /* element in meas */
                xflag2 = 0;
                while ( xflag2 == 0 )
                    {
                    bitter = nbuff6[ mctr + nctr ];
                    switch ( bitter )
                    {
                    case 0x00:
                        xflag2 = 1; /* meas done */
                        break;
                    default:
                        test = toint ( bitter );
                        if ( test >= 128 )
                            {
                                    nctr++;
                                    }
                                    else
                                    {
                                    ftab3[ ctr ] = test;
                                    nctr++;
                                    ctr++;
                                    }
                                break;
                            }
                            }
                }
                mctr += 24;
                }
            }

/*
*   convert measure freq data
*   to sequencer duration data
*/

cvd3()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1, tempd1;
    xflag1 = 0;
    ctr = 0;    /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff5[ mctr ];
        switch ( c1 )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff5[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    tempd1 = toint ( bitter );
                    test = dequ ( tempd1 );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        switch ( test )
                        {
                        case    100:
                        case    117:
                            ctr--;  /* previous */
                            il1=dtab3[ctr]; /*dur*/
                            ctr++;
                            dtab3[ctr] = il1/2;
                            ctr++; /*restore dur*/
                            nctr++; /*new nbuff*/
                            break;
                        default:
                            dtab3[ ctr ] = test;
                            nctr++;
                            ctr++;
                            break;
                        }
                        }
                    break;
                }
                }
        }
        mctr += 24;
        }
    }

/*
*   call to convert editor buffs to
*   sequencer buffs
*/

conf4()
    {
    char    cc1;
    cc1 = nbuff8[ 0 ];
    switch ( cc1 )
        {
        case 0x00:
            fflag1 = 0;
            break;
        default:
            fflag1 = 1;
            cvf4();
            cvd4();
            break;
        }
    }

/*
*   convert measure freq data
*   to sequencer freq data
*/

cvf4()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1;
    xflag1 = 0;
    ctr = 0;     /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff8[ mctr ];
        switch ( c1 )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;    /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff8[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    test = toint ( bitter );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        ftab4[ ctr ] = test;
                        nctr++;
                        ctr++;
                        }
                    break;
                }
                }
        }
        mctr += 24;
        }
    }

/*
*   convert measure freq data
*   to sequencer duration data
*/

cvd4()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1, tempd1;
    xflag1 = 0;
    ctr = 0;    /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff7[ mctr ];
        switch ( c1 )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff7[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    tempdl = toint ( bitter );
                    test = dequ ( tempdl );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        switch ( test )
                        {
                        case    100:
                        case    117:
                            ctr--;  /* previous */
                            ill=dtab4[ctr]; /*dur*/
                            ctr++;
                            dtab4[ctr] = ill/2;
                            ctr++; /*restore dur*/
                            nctr++; /*new nbuff*/
                            break;
                        default:
                            dtab4[ ctr ] = test;
                            nctr++;
                            ctr++;
                            break;
                        }
                        }
                    break;
                }
                }
        }
        mctr += 24;
        }
    }

/*
*   call to convert editor buffs to
*   sequencer buffs
*/
```

```
conf5()
    {
    char    ccl;
    ccl = nbuff10[ 0 ];
    switch ( ccl )
        {
        case 0x00:
            fflag1 = 0;
            break;
        default:
            fflag1 = 1;
            cvf5();
            cvd5();
            break;
        }
    }

/*
*   convert measure freq data
*   to sequencer freq data
*/

cvf5()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1;
    xflag1 = 0;
    ctr = 0;    /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff10[ mctr ];
        switch ( c1 )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff10[ mctr + nctr ];
                switch ( bitter )
                {
```

```
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    test = toint ( bitter );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        ftab5[ ctr ] = test;
                        nctr++;
                        ctr++;
                        }
                    break;
                }
                }
            }
            mctr += 24;
            }
        }

/*
*   convert measure freq data
*   to sequencer duration data
*/

cvd5()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1, tempd1;
    xflag1 = 0;
    ctr = 0;     /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff9[ mctr ];
        switch ( c1 )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff9[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    tempdl = toint ( bitter );
                    test = dequ ( tempdl );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        switch ( test )
                        {
                        case    100:
                        case    117:
                            ctr--;  /* previous */
                            il1=dtab5[ctr]; /*dur*/
                            ctr++;
                            dtab5[ctr] = il1/2;
                            ctr++; /*restore dur*/
                            nctr++; /*new nbuff*/
                            break;
                        default:
                            dtab5[ ctr ] = test;
                            nctr++;
                            ctr++;
                            break;
                        }
                        }
                     break;
                }
                }
        }
        mctr += 24;
        }
    }
```

```
/*
*   call to convert editor buffs to
*   sequencer buffs
*/

conf6()
    {
    char    ccl;
    ccl = nbuff12[ 0 ];
    switch ( ccl )
        {
        case 0x00:
            fflag1 = 0;
            break;
        default:
            fflag1 = 1;
            cvf6();
            cvd6();
            break;
        }
    }

/*
*   convert measure freq data
*   to sequencer freq data
*/

cvf6()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1;
    xflag1 = 0;
    ctr = 0;    /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff12[ mctr ];
        switch ( c1 )
        {
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff12[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    test = toint ( bitter );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        ftab6[ ctr ] = test;
                        nctr++;
                        ctr++;
                        }
                    break;
                }
                }
        }
        mctr += 24;
        }
    }

/*
*   convert measure freq data
*   to sequencer duration data
*/

cvd6()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1, tempd1;
    xflag1 = 0;
    ctr = 0;     /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff11[ mctr ];
        switch ( c1 )
        {
```

```
case 0x00:
    xflag1 = 1;
    break;
default:
    nctr = 0;   /* element in meas */
    xflag2 = 0;
    while ( xflag2 == 0 )
        {
        bitter = nbuff11[ mctr + nctr ];
        switch ( bitter )
        {
        case 0x00:
            xflag2 = 1; /* meas done */
            break;
        default:
            tempd1 = toint ( bitter );
            test = dequ ( tempd1 );
            if ( test >= 128 )
                {
                nctr++;
                }
                else
                {
                switch ( test )
                {
                case    100:
                case    117:
                    ctr--;  /* previous */
                    il1=dtab6[ctr]; /*dur*/
                    ctr++;
                    dtab6[ctr] = il1/2;
                    ctr++; /*restore dur*/
                    nctr++; /*new nbuff*/
                    break;
                default:
                    dtab6[ ctr ] = test;
                    nctr++;
                    ctr++;
                    break;
                }
                }
            break;
        }
        }
}
mctr += 24;
```

```
          }
     }

/*
*    call to convert editor buffs to
*    sequencer buffs
*/

conf7()
     {
     char    cc1;
     cc1 = nbuff14[ 0 ];
     switch ( cc1 )
          {
          case 0x00:
               fflag1 = 0;
               break;
          default:
               fflag1 = 1;
               cvf7();
               cvd7();
               break;
          }
     }

/*
*    convert measure freq data
*    to sequencer freq data
*/

cvf7()
     {
     char    bitter, c1;
     int mctr, ctr, nctr, xflag1, xflag2, test, il1;
     xflag1 = 0;
     ctr = 0;     /* ftab element # */
     mctr = 0;
     mctr = ( st_m - 1 ) * 24;
     while ( xflag1 == 0 )
          {
          c1 = nbuff14[ mctr ];
          switch ( c1 )
          {
          case 0x00:
               xflag1 = 1;
               break;
```

```
        default:
            nctr = 0;    /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff14[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    test = toint ( bitter );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        ftab7[ ctr ] = test;
                        nctr++;
                        ctr++;
                        }
                    break;
                }
                }
        }
        mctr += 24;
        }
    }

/*
*   convert measure freq data
*   to sequencer duration data
*/

cvd7()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1, tempd1;
    xflag1 = 0;
    ctr = 0;    /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
```

```
    cl = nbuff13[ mctr ];
    switch ( cl )
    {
    case 0x00:
        xflag1 = 1;
        break;
    default:
        nctr = 0;   /* element in meas */
        xflag2 = 0;
        while ( xflag2 == 0 )
            {
            bitter = nbuff13[ mctr + nctr ];
            switch ( bitter )
            {
            case 0x00:
                xflag2 = 1; /* meas done */
                break;
            default:
                tempd1 = toint ( bitter );
                test = dequ ( tempd1 );
                if ( test >= 128 )
                    {
                    nctr++;
                    }
                    else
                    {
                    switch ( test )
                    {
                    case    100:
                    case    117:
                        ctr--;  /* previous */
                        il1=dtab7[ctr]; /*dur*/
                        ctr++;
                        dtab7[ctr] = il1/2;
                        ctr++; /*restore dur*/
                        nctr++; /*new nbuff*/
                        break;
                    default:
                        dtab7[ ctr ] = test;
                        nctr++;
                        ctr++;
                        break;
                    }
                    }
                break;
            }
```

```
                    }
                }
                mctr += 24;
                }
            }

/*
*   call to convert editor buffs to
*   sequencer buffs
*/

conf8()
    {
    char    cc1;
    cc1 = nbuff16[ 0 ];
    switch ( cc1 )
        {
        case 0x00:
            fflag1 = 0;
            break;
        default:
            fflag1 = 1;
            cvf8();
            cvd8();
            break;
        }
    }

/*
*   convert measure freq data
*   to sequencer freq data
*/

cvf8()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1;
    xflag1 = 0;
    ctr = 0;     /* ftab element # */
    mctr = 0;
    mctr = ( st_m - 1 ) * 24;
    while ( xflag1 == 0 )
        {
        c1 = nbuff16[ mctr ];
        switch ( c1 )
        {
```

```
        case 0x00:
            xflag1 = 1;
            break;
        default:
            nctr = 0;   /* element in meas */
            xflag2 = 0;
            while ( xflag2 == 0 )
                {
                bitter = nbuff16[ mctr + nctr ];
                switch ( bitter )
                {
                case 0x00:
                    xflag2 = 1; /* meas done */
                    break;
                default:
                    test = toint ( bitter );
                    if ( test >= 128 )
                        {
                        nctr++;
                        }
                        else
                        {
                        ftab8[ ctr ] = test;
                        nctr++;
                        ctr++;
                        }
                    break;
                }
                }
        }
        mctr += 24;
        }
    }

/*
*   convert measure freq data
*   to sequencer duration data
*/

cvd8()
    {
    char    bitter, c1;
    int mctr, ctr, nctr, xflag1, xflag2, test, il1, tempd1;
    xflag1 = 0;
    ctr = 0;    /* ftab element # */
    mctr = 0;
mctr = ( st_m - 1 ) * 24;
while ( xflag1 == 0 )
    {
    c1 = nbuff15[ mctr ];
    switch ( c1 )
    {
    case 0x00:
        xflag1 = 1;
        break;
    default:
        nctr = 0;   /* element in meas */
        xflag2 = 0;
        while ( xflag2 == 0 )
            {
            bitter = nbuff15[ mctr + nctr ];
            switch ( bitter )
            {
            case 0x00:
                xflag2 = 1; /* meas done */
                break;
            default:
                tempd1 = toint ( bitter );
                test = dequ ( tempd1 );
                if ( test >= 128 )
                    {
                    nctr++;
                    }
                    else
                    {
                    switch ( test )
                    {
                    case    100:
                    case    117:
                        ctr--;  /* previous */
                        il1=dtab8[ctr]; /*dur*/
                        ctr++;
                        dtab8[ctr] = il1/2;
                        ctr++; /*restore dur*/
                        nctr++; /*new nbuff*/
                        break;
                    default:
                        dtab8[ ctr ] = test;
                        nctr++;
                        ctr++;
                        break;
                    }
```

```
                            }
                        break;
                    }
                    }
            }
            mctr += 24;
            }
    }



/*
**  end of file interf.c
*/
```

4.7 `move1.c` source code

The routines in this file are used to move animated shapes about the screen. Before we present the code to move the shapes, it is important that we describe the process used to create shapes. We have defined a grid of 16 pixels wide by 12 pixels high for the shape. The algorithm which we used dictates that there will be one animated shape for each voice that plays. When a voice is at rest, the animated shape will disappear. The shape's horizontal position on the screen represents the note value. For example, the very lowest 'C' note will appear at the left of the display and the very highest 'B' note will appear at the far right of the display. If the note playing on voice 1 changes from, say middle 'C' to 'C#', the shape will move ever so slightly to the right. Those of you familiar with the PLAYERS and MISSILES from the Atari 130XE will appreciate their power. The Atari ST does not have PLAYERS or sprites available for programmers. The animation code is very similar to the code presented in the `'btdata.c'` file section that writes an icon to the screen. The only difference is that the shapes described in the three dimensional array are erased, and then re-written at varying positions on the screen. The grid for the shapes is two bytes wide and 12 bytes high. Each bit that is on will represent a colored pixel being displayed on the screen. Since we want the shape to move smoothly about the screen we need 16 different shapes which may be drawn to the screen.

Shape for GRID

GRID 1

row	byte 1 bit 7	6	5	4	3	2	1	0	byte 2 bit 7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GRID 2

row	byte 1 bit 7	6	5	4	3	2	1	0	byte 2 bit 7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

It is not necessary to show all the grids here because the column of ones will continue to move to the right with each of the subsequent grids. The animation algorithm of the line moving to the right would be coded with the following logic:

1) Print the bytes for GRID 1
2) DELAY
3) Erase GRID 1
4) Print the bytes for GRID 2
5) DELAY
6) erase GRID 2
7) ...etc...

Recalling that we are using the medium resolution color mode and taking the bit planing arrangement of the Atari ST into consideration, it is clear that we need to print 4 bytes across in order to have the line shape appear as black. The screen byte arrangement for GRID 1 & 2 would appear as follows. Note that the binary number representations have been replaced by their decimal counterparts.

SCREEN BYTE ARRANGEMENTS

GRID 1

byte 1	byte 2	byte 3	byte 4
128	0	128	0
128	0	128	0
128	0	128	0
128	0	128	0
128	0	128	0
128	0	128	0
128	0	128	0
128	0	128	0
128	0	128	0
128	0	128	0
128	0	128	0
128	0	128	0

SCREEN BYTE ARRANGEMENTS			
GRID 2			
byte 1	byte 2	byte 3	byte 4
64	0	64	0
64	0	64	0
64	0	64	0
64	0	64	0
64	0	64	0
64	0	64	0
64	0	64	0
64	0	64	0
64	0	64	0
64	0	64	0
64	0	64	0
64	0	64	0

Here is the source to 'move1.c'.

```
/*
**  move1.c file
*/

extern  int octsw0,octsw1,octsw2,octsw3,octsw4,octsw5,octsw6,octsw7;
extern  char *ptr1, *ptr2, *ptr3,*ptr4,*ptr5,*ptr6,*ptr7,*ptr8;

char    *newloc, *new2loc, *new3loc,*new4loc,*new5loc,*new6loc,*new7loc;
char    *new8loc;
extern  long    screen1;

/*
**  animate shape tables
*/

char    annote[16][12][4] = {

/*
**  grid 1
*/
```

```
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,
    128,0,128,0,

/*
**  grid 2
*/
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,
    64,0,64,0,

/*
**  grid 3
*/

    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
    32,0,32,0,
/*
**   grid 4
*/
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,
     16,0,16,0,

/*
**   grid 5
*/
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,
     8,0,8,0,

/*
**   grid 6
*/
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
     4,0,4,0,
/*
**   grid 7
*/
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,
     2,0,2,0,

/*
**   grid 8
*/
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,
     1,0,1,0,

/*
**   grid 9
*/
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
     0,128,0,128,
/*
**   grid 10
*/
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,
     0,64,0,64,

/*
**   grid 11
*/
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,
     0,32,0,32,

/*
**   grid 12
*/
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
     0,16,0,16,
/*
**  grid 13
*/
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,
    0,8,0,8,

/*
**  grid 14
*/
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,
    0,4,0,4,

/*
**  grid 15
*/
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
    0,2,0,2,
/*
**  grid 16
*/
    0,1,0,1,
    0,1,0,1,
    0,1,0,1,
    0,1,0,1,
    0,1,0,1,
    0,1,0,1,
    0,1,0,1,
    0,1,0,1,
    0,1,0,1,
    0,1,0,1,
    0,1,0,1,
    0,1,0,1
}


/*-----move shape as voice 1 moves 0-----*/

/*
**  This function receives the note #.  It will
**  range between 36 to 96. Here is the animation algorithm.
**  1) take received number and times by 4
**  2) divide by 16 and store in 'quot'
**  3) remainder goes into 'rem'
**  4) new screen location goes into 'newloc'
**  5) ptr1 holds OLD location and erases old shape
**  6) new shape (determined by 'rem' value)
**     The 'rem' value calls the appropriate
**     grid pattern to be stuffed to the screen.
**  7) ptr1 is reset to newest shape position.
**  All the voice animation follows the identical
**  algorithm .
**
*/


movel1 ( num1 )
    int num1;
    {
    register    int lx, ly;
    int     bflag, quot, rem;
    bflag = num1;
    switch ( octsw0 )
        {
        case 0x01:
            num1 += 12;
            break;
        case 0x02:
            num1 += 24;
            break;
        default:
            break;
        }
    num1 *= 4;                              /* times 2 */
    quot = num1 / 16;
    rem = num1 % 16;                        /* pixel offset */
    newloc = screen1 + ( 4 * quot );        /* byte offset left */

    for ( ly = 0; ly < 12; ly++ )           /* erase shape */
        for ( lx = 0; lx < 4; lx++ )
            *( ptr1 + 4 + ( 74 * 160 ) + ( ly * 160 ) + lx ) = 0;

    switch ( bflag )
        {
        case 0x01:
            break;
        default:
        for ( ly = 0; ly < 12; ly++ )    /* draw shape */
            for ( lx = 0; lx < 4; lx++ )
            *(newloc + 4 + ( 74 * 160 ) + ( ly * 160 ) + lx ) =
                        annote[rem][ly][lx];
            break;
        }
    ptr1 = newloc;
    }

/*-----move the right voice 1 shape -----*/


move21 ( num2 )
    int num2;
    {
    register    int lx, ly;
    int quot, rem, bflag;
    bflag = num2;
    switch ( octsw1 )
        {
        case 0x01:
            num2 += 12;
            break;
        case 0x02:
            num2 += 24;
```

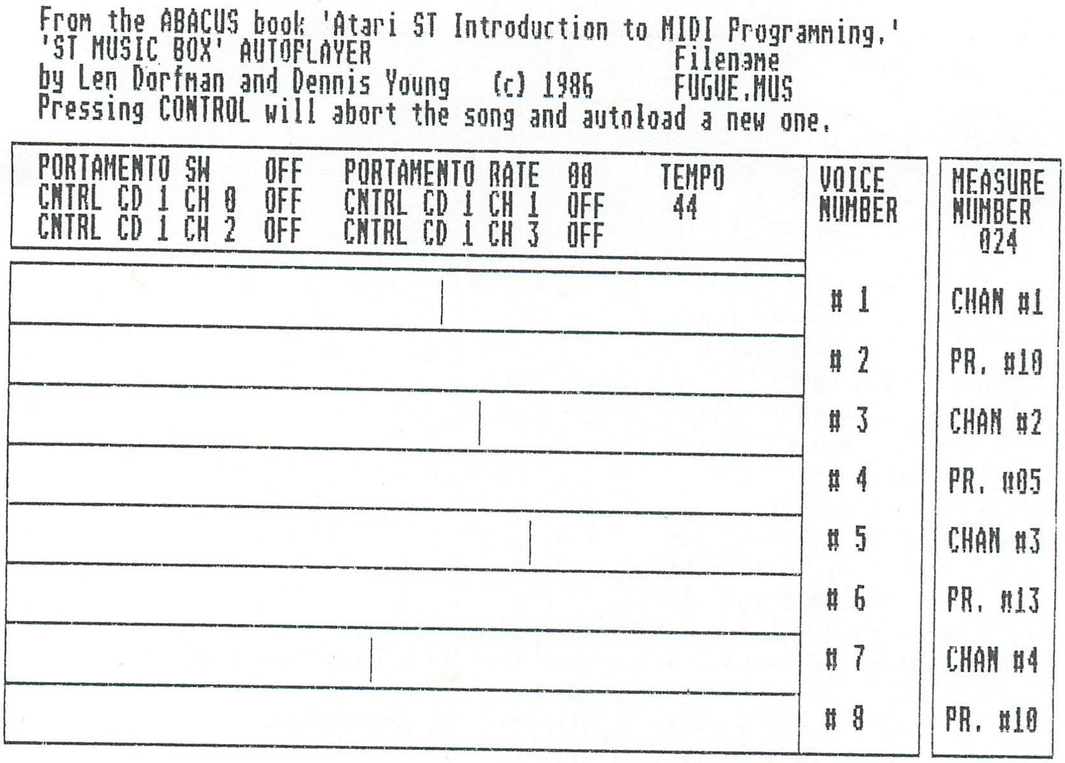
```
                break;
            default:
                break;
            }
        num2 *= 4;                    /* times 2 */
        quot = num2 / 16;
        rem = num2 % 16;              /* pixel offset */
        new2loc = screen1 + ( 4 * quot );    /* byte offset left */

        for ( ly = 0; ly < 12; ly++ )        /* erase shape */
            for ( lx = 0; lx < 4; lx++ )
                *( ptr2 + 4 + ( 90 * 160 ) + ( ly * 160 ) + lx ) = 0;

        switch ( bflag )
            {
            case 0x01:
                break;
            default:
            for ( ly = 0; ly < 12; ly++ )    /* draw shape */
                for ( lx = 0; lx < 4; lx++ )
                *(new2loc + 4 + ( 90 * 160 ) + ( ly * 160 ) + lx ) =
                               annote[rem][ly][lx];
            break;
            }
        ptr2 = new2loc;
        }

/*-----move the right voice 3 shape -----*/


move31 ( num3 )
        int num3;
        {
        register    int lx, ly;
        int bflag, quot, rem;
        bflag = num3;
        switch ( octsw2 )
            {
            case 0x01:
                num3 += 12;
                break;
            case 0x02:
                num3 += 24;
                break;
            default:
                break;
            }
```

```
    num3 *= 4;                             /* times 2 */
    quot = num3 / 16;
    rem = num3 % 16;                       /* pixel offset */
    new3loc = screen1 + ( 4 * quot );      /* byte offset left */

    for ( ly = 0; ly < 12; ly++ )          /* erase shape */
        for ( lx = 0; lx < 4; lx++ )
            *( ptr3 + 4 + ( 106 * 160 ) + ( ly * 160 ) + lx ) = 0;

    switch ( bflag )
        {
        case 0x01:
            break;
        default:
        for ( ly = 0; ly < 12; ly++ )      /* draw shape */
            for ( lx = 0; lx < 4; lx++ )
            *(new3loc + 4 + ( 106 * 160 ) + ( ly * 160 ) + lx ) =
                                annote[rem][ly][lx];

            break;
        }
    ptr3 = new3loc;
    }


/*-----move the right voice 4 shape -----*/


move4l ( num4 )
    int num4;
    {
    register    int lx, ly;
    int bflag, quot, rem;
    bflag = num4;
    switch ( octsw3 )
        {
        case 0x01:
            num4 += 12;
            break;
        case 0x02:
            num4 += 24;
            break;
        default:
            break;
        }
    num4 *= 4;          /* times 2 */
    quot = num4 / 16;
    rem = num4 % 16;              /* pixel offset */
    new4loc = screenl + ( 4 * quot );     /* byte offset left */

    for ( ly = 0; ly < 12; ly++ )         /* erase shape */
        for ( lx = 0; lx < 4; lx++ )
            *( ptr4 + 4 + ( 122 * 160 ) + ( ly * 160 ) + lx ) = 0;

    switch ( bflag )
        {
        case 0x01:
            break;
        default:
        for ( ly = 0; ly < 12; ly++ )
    /* draw shape */
            for ( lx = 0; lx < 4; lx++ )
            *(new4loc + 4 + ( 122 * 160 ) + ( ly * 160 ) + lx ) =
                                annote[rem][ly][lx];
            break;
        }
    ptr4 = new4loc;
    }

/*-----move the right voice 5 shape -----*/


move5l ( num5 )
    int num5;
    {
    register    int lx, ly;
    int bflag, quot, rem;
    bflag = num5;
    switch ( octsw4 )
        {
        case 0x01:
            num5 += 12;
            break;
        case 0x02:
            num5 += 24;
            break;
        default:
            break;
        }
    num5 *= 4;        /* times 2 */
    quot = num5 / 16;
    rem = num5 % 16;              /* pixel offset */
    new5loc = screenl + ( 4 * quot );    /* byte offset left */

    for ( ly = 0; ly < 12; ly++ )        /* erase shape */
```

```
        for ( lx = 0; lx < 4; lx++ )
            *( ptr5 + 4 + ( 138 * 160 ) + ( ly * 160 ) + lx ) = 0;

    switch ( bflag )
        {
        case 0x01:
            break;
        default:
        for ( ly = 0; ly < 12; ly++ )    /* draw shape */
            for ( lx = 0; lx < 4; lx++ )
            *(new5loc + 4 + ( 138 * 160 ) + ( ly * 160 ) + lx ) =
                                annote[rem][ly][lx];

            break;
        }
    ptr5 = new5loc;
    }

/*-----move the right voice 6 shape -----*/


move6l ( num6 )
    int num6;
    {
    register    int lx, ly;
    int bflag, quot, rem;
    bflag = num6;
    switch ( octsw5 )
        {
        case 0x01:
            num6 += 12;
            break;
        case 0x02:
            num6 += 24;
            break;
        default:
            break;
        }
    num6 *= 4;        /* times 2 */
    quot = num6 / 16;
    rem = num6 % 16;               /* pixel offset */
    new6loc = screen1 + ( 4 * quot );    /* byte offset left */

    for ( ly = 0; ly < 12; ly++ )         /* erase shape */
        for ( lx = 0; lx < 4; lx++ )
            *( ptr6 + 4 + ( 154 * 160 ) + ( ly * 160 ) + lx ) = 0;

    switch ( bflag )
```

```
        {
        case 0x01:
            break;
        default:
        for ( ly = 0; ly < 12; ly++ )    /* draw shape */
            for ( lx = 0; lx < 4; lx++ )
            *(new6loc + 4 + ( 154 * 160 ) + ( ly * 160 ) + lx ) =
                                annote[rem][ly][lx];
            break;
        }
    ptr6 = new6loc;
    }

/*-----move the right voice 7 shape -----*/


move7l ( num7 )
    int num7;
    {
    register    int lx, ly;
    int bflag, quot, rem;
    bflag = num7;
    switch ( octsw6 )
        {
        case 0x01:
            num7 += 12;
            break;
        case 0x02:
            num7 += 24;
            break;
        default:
            break;
        }
    num7 *= 4;      /* times 2 */
    quot = num7 / 16;
    rem = num7 % 16;            /* pixel offset */
    new7loc = screen1 + ( 4 * quot );    /* byte offset left */

    for ( ly = 0; ly < 12; ly++ )        /* erase shape */
        for ( lx = 0; lx < 4; lx++ )
            *( ptr7 + 4 + ( 170 * 160 ) + ( ly * 160 ) + lx ) = 0;

    switch ( bflag )
        {
        case 0x01:
            break;
        default:
```

```
        for ( ly = 0; ly < 12; ly++ )    /* draw shape */
            for ( lx = 0; lx < 4; lx++ )
            *(new7loc + 4 + ( 170 * 160 ) + ( ly * 160 ) + lx ) =
                                annote[rem][ly][lx];
            break;
        }
    ptr7 = new7loc;
    }

/*-----move the right voice 8 shape -----*/


move8l ( num8 )
    int num8;
    {
    register    int lx, ly;
    int bflag, quot, rem;
    bflag = num8;
    switch ( octsw7 )
        {
        case 0x01:
            num8 += 12;
            break;
        case 0x02:
            num8 += 24;
            break;
        default:
            break;
        }
    num8 *= 4;       /* times 2 */
    quot = num8 / 16;
    rem = num8 % 16;               /* pixel offset */
    new8loc = screenl + ( 4 * quot );    /* byte offset left */

    for ( ly = 0; ly < 12; ly++ )        /* erase shape */
        for ( lx = 0; lx < 4; lx++ )
            *( ptr8 + 4 + ( 186 * 160 ) + ( ly * 160 ) + lx ) = 0;

    switch ( bflag )
        {
        case 0x01:
            break;
        default:
        for ( ly = 0; ly < 12; ly++ )    /* draw shape */
            for ( lx = 0; lx < 4; lx++ )
            *(new8loc + 4 + ( 186 * 160 ) + ( ly * 160 ) + lx ) =
                                annote[rem][ly][lx];
            break;
        }
    ptr8 = new8loc;
    }


/*
**  end of move1.c
*/
```



```
From the ABACUS book 'Atari ST Introduction to MIDI Programming.'
'ST MUSIC BOX' AUTOPLAYER                               Filename
by Len Dorfman and Dennis Young    (c) 1986             FUGUE.MUS
Pressing CONTROL will abort the song and autoload a new one.

PORTAMENTO SW    OFF   PORTAMENTO RATE  00     TEMPO     VOICE     MEASURE
CNTRL CD 1 CH 0  OFF   CNTRL CD 1 CH 1  OFF     44       NUMBER    NUMBER
CNTRL CD 1 CH 2  OFF   CNTRL CD 1 CH 3  OFF                         024
                                                         # 1       CHAN #1
                                                         # 2       PR. #10
                                                         # 3       CHAN #2
                                                         # 4       PR. #05
                                                         # 5       CHAN #3
                                                         # 6       PR. #13
                                                         # 7       CHAN #4
                                                         # 8       PR. #10
```

Appendices

Compiling The Programs

The following batch files were used to compile the `scaler.c`, `readkey.c` and `patch.c` source code. Rember when using the Acylon C compiler that getchar expects a control z to end the input.

C compiler batch file, `C.BAT`:

```
cp68 %1.c %1.i
c068 %1.i %1.1 %1.2 %1.3 -f
rm %1
c168 %1.1 %1.2 %1.s
rm %1.1
rm %1.2
as68 -l -u %1.s
rm %1.s
wait
```

Link file, `LINK1.BAT`:

```
link68 [u] %1.68k=gemstart,%1,vdibind,osbind,aesbind,gemlib,libf
relmod %1
rm %1.68k
wait
```

The files `eio.c`, `mdrive.c`, `sets.c`, `btdata.c`, `interf.c` and `move1.c` were first compiled using the above C.BAT file, then linked with the following file. The first two lines are entered as one complete line.

```
link68 [u] %1.68k=gemstart,eio,mdrive,sets,btdata,interf,move1,vdibind,
osbind,aesbind,gemlib,libf
relmod %1
rm %1.68k
wait
```

Optional Diskette notes.

The optional diskette contains the soucrce code and compiled programs for all the programs presented in the book. It also includes the following music files:

```
RAGS.MUS
WILDR.MUS
NANETTE.MUS
MINUET.MUS
FUGE.MUS
```

The optional diskette also includes two version of `btdata.c,` the one presented in the book and one that contains alnernate animation. The main focus of the book was MIDI programming and it was not within the scope of the book to provide the source for the alternate animation routines.

Index

ACTIVE SENSING MESSAGE 46
AUTO-PLAYER (ST Music Box) 7,93-250

bconin 57-58
bconout 56-58
BIOS 56-59
Bits and bytes 12,31-35
btdata.c 168-192
bulk dump 47
CONTINUE MESSAGE 46
channels 16

data bytes 37
Data format 36
DIN cables 11
Duration 4-5,26

eio.c 101-118
EOX MESSAGE 48
Extended BIOS 56-59

Hexadecimal notation 53-55

interf.c 193-232
largo 5

manufacture's ID numbers 49
mdrive.c 119-144
MESSAGE TYPES 36,46
MIDI
- beat 44
- clock 44
- IMPLEMENTATION 13-20,23
- implementation chart 24
- IN/OUT 3,11
- language 8,13-20,31-49
- software 26-28
- SPECIFICATION 1.0 13-20

standard	13
MODES	17-18
`movel.c`	233-250
Note values--see Duration	
OCTAL	53
OMINI/POLY	17
OMNI ON.MONO	17
OMNI OFF/POLY	17
`patch.c`	77-92
patches	19
Parallel transmission	12
Pitch	6-7
pitch bend	16
Portamento	4
RAM cartridge	21
`readkey.c`	66-76
Real Time Capture	27,66
`scaler.c`	58,59-65
Serial transmission	12
`set.c`	145-167
SONG SELECT	44
SONG POINTER	44
ST MUSIC BOX	i,7,95-250
Status bytes	37
System exclusive	37,47
System real time messages	45
System reset message	46
Tempo	5,26
Vibrato	16
wave form	20

Optional Diskette

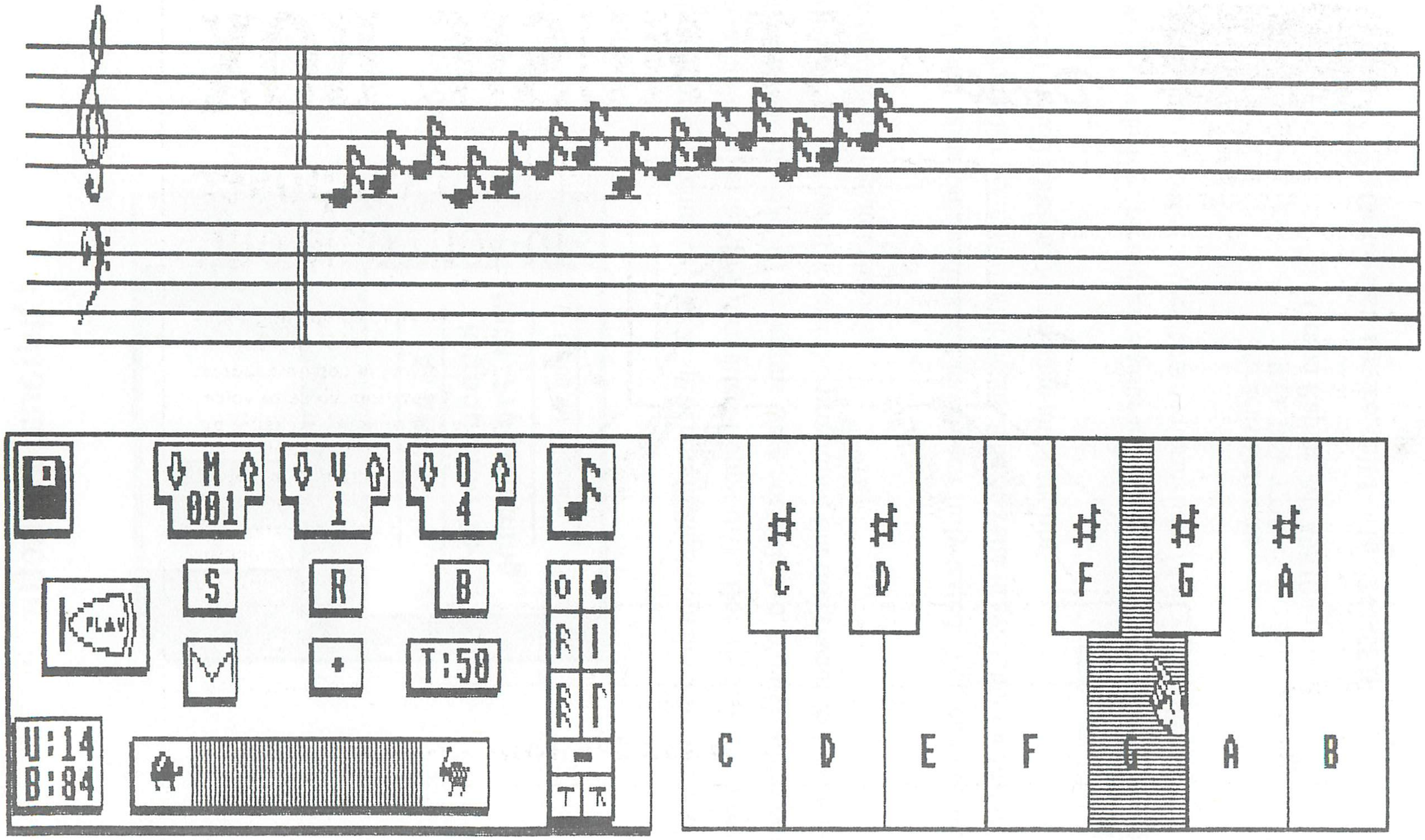
For your convenience, the 'C' program listings contained in this book are available on an SF354 formatted floppy disk. Due to diskette directory limitations the BASIC programs were not included. You should order the diskette if you want to use the programs, but don't want to type them in from the listings in the book.

All programs on the diskette have been fully tested. You can change the programs for your particular needs. The diskette is available for $14.95 plus $2.00 ($5.00 foreign) for postage and handling.

When ordering, please give your name and shipping address. Enclose a check, money order or credit card information. Mail your order to:

Abacus Software
P.O. Box 7219
Grand Rapids, MI 49510

Or for fast service, call **1- 616 / 241-5510**.


M 001
V 1
O 4
S
R
B
T:50
PLAY
U:14
B:84
C
D
E
F
G
A
B

XLEnt Software

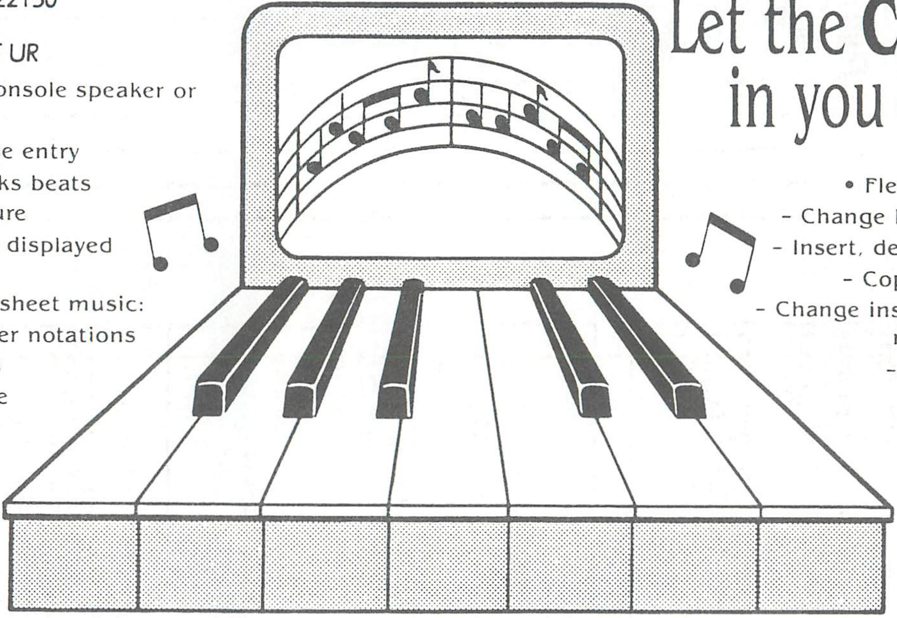
P.O. Box 5228
Springfield, Virginia 22150
(703) 644-8881
Telex 269728 XLNT UR

Presents **ST MUSIC BOX**

copyright 1986

Let the **Composer** in you come out.

- Compose for ST console speaker or MIDI synthesizer
- Keyboard or mouse entry
- Smart editor checks beats entered per measure
- All parameter info displayed during playback
- Print high quality sheet music:
  - Add lyrics & other notations
  - Include graphics
  - Print single voice or combination
- Conforms to MIDI standard
- Supports monochrome and color



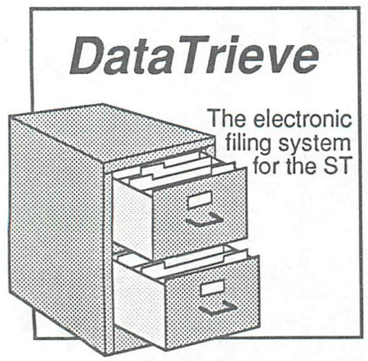
- Flexible composing tool:
  - Change key & time signatures
  - Insert, delete & copy measures
  - Copy from voice to voice
  - Change instruments measure by measure for any voice
  - Alter tempo measure by measure
  - Control synthesizer portamento
  - Capability to transpose notes
  - Load/Save rhythms, single voice or compositions

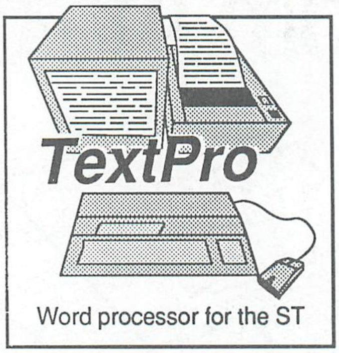
$49.95

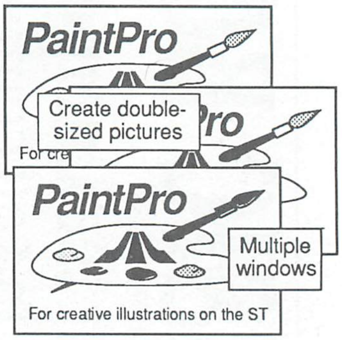
By Dennis Young & Len Dorfman

Includes four compositions!

AA Rated Software Atari and Abacus



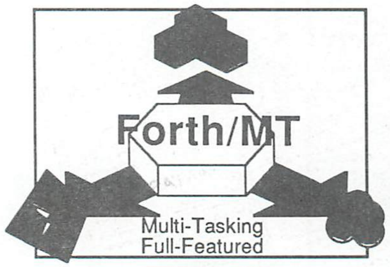


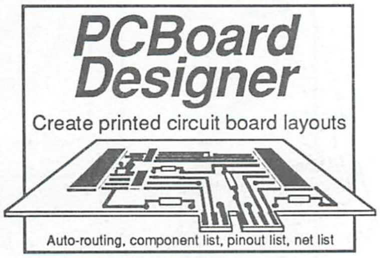

PaintPro
Create double-sized pictures
PaintPro
Multiple windows
For creative illustrations on the ST

ST DataTrieve
A simple-to-use and versatile database manager. Features help screens; lightning-fast operation; tailorable display using multiple fonts; user-definable edit masks; capacity up to 64,000 records. Supports multiple files. RAM-disk support for 1040ST. Complete search, sort and file subsetting. Interfaces to TextPro. Easy printer control. **$49.95**

ST TextPro
Wordprocessor with professional features and easy-to-use! Full-screen editing with mouse or keyboard shortcuts. High speed input, scrolling and editing; sideways printing; multi-column output; flexible printer installation; automatic index and table of contents; up to 180 chars/line; 30 definable function keys; metafile output; much more. **$49.95**

ST PaintPro
A GEM™ among ST drawing programs. Very friendly, but very powerful design and painting program. A *must* for everyone's artistic or graphics needs. Use up to **three** windows. You can even cut & paste between windows. Free-form sketching; lines, circles, ellipses, boxes, text, fill, copy, move, zoom, spray, paint, erase, undo, help. Double-sized picture format. **$49.95**




PCBoard Designer
Create printed circuit board layouts
Auto-routing, component list, pinout list, net list

ST Forth/MT
Powerful, multi-tasking Forth for the ST. A complete, 32-bit implementation based on Forth-83 standard. Development aids: full screen editor, monitor, macro assembler. 1500+ word library. TOS/LINEA commands. Floating point and complex arithmetic. **$49.95**

ST AssemPro
Professional developer's package includes editor, two-pass interactive assembler with error locator, online help including instruction address mode and GEM parameter information, monitor-debugger, disassembler and 68020 simulator, more. **$59.95**

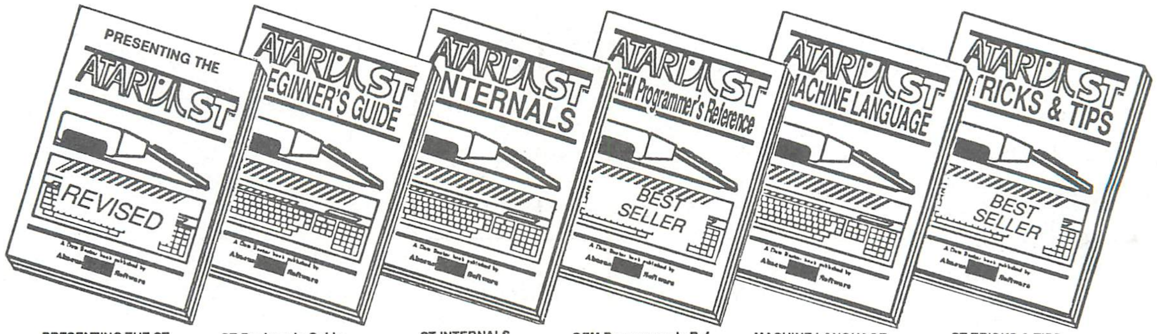
PCBoard Designer
Interactive, computer aided design package that automates layout of printed circuit boards. **Auto-routing**, 45° or 90° traces; two-sided boards; pin-to-pin, pin-to-BUS or BUS-to-BUS. Rubber-banding of components during placement. Outputs pinout, component and net list. **$395.00**

ST and 1040ST are trademarks of Atari Corp.
GEM is a trademark of Digital Research Inc.

Call **now** for the name of the dealer nearest you. Or order directly using your MC, Visa or Amex card. Add $4.00 per order for shipping. Foreign orders add $10.00 per item. Call (616) 241-5510 or write for your **free** catalog. 30-day money back software guarantee. Dealers inquires welcome–over 1400 dealers nationwide.

Abacus
P.O. Box 7219 Dept.NB Grand Rapids, MI 49510
Phone 616/241-5510 • Telex 709-101 • Fax 616/241-5021

Top shelf books from Abacus



PRESENTING THE ST
Gives you an in-depth look at this sensational new computer. Discusses the architecture of the ST, working with GEM, the mouse, operating system, all the various interfaces, the 68000 chip and its instructions, LOGO. 200pp $16.95

ST Beginner's Guide
Written for the firsthand ST user. Get a basic understanding of your ST. Explore LOGO and BASIC from the ground up. Simple explanations of the hardware and internal workings of the ST. Illustrations, diagrams. Glossary. Index. 200pp $14.95

ST INTERNALS
Essential guide to the inside information of the ST. Detailed descriptions of sound and graphics chips, internal hardware, I/O ports, using GEM. Commented BIOS listing. An indispensible reference for your ST library. 450pp $19.95

GEM Programmer's Ref.
For serious programmers needing detailed information on GEM. Presented in an easy-to-understand format. All examples are in C and assembly language. Covers VDI and AES functions. No serious programer should be without. 410pp $19.95

MACHINE LANGUAGE
Program in the fastest language for your ATARI ST. Learn 68000 assembly language, its numbering system, use of registers, structure & important details of instruction set, and use of internal system routines. Geared for the ST. 280pp $19.95

ST TRICKS & TIPS
Fantastic collection of programs and info for the ST. Complete programs include: super-fast RAM disk; time-saving printer spooler; color print hardcopy; plotter output hardcopy; creating accessories. Money saving tricks and tips. 260pp $19.95

ST GRAPHICS & SOUND
Detailed guide to graphics and sound on the ST. 2D & 3D function plotters, Moiré patterns, graphic memory and various resolutions, fractals, recursion, waveform generation. Examples written in C, LOGO, BASIC and Modula2. 250pp $19.95

ST LOGO GUIDE
Take control of your ST by learning ST LOGO—the easy to use, powerful language. Topics include: file handling, recursion-Hilbert & Sierpinski curves, 2D and 3D function plots, data structure, error handling. Helpful guide for ST LOGO users. $19.95

ST PEEKS & POKES
Enhance your programs with the examples found within this book. Explores using different languages BASIC, C, LOGO and machine language, using various interfaces, memory usage, reading and saving from and to disk, more. 280pp $16.95

BASIC Training Guide
Thorough guide for learning ST BASIC programming. Detailed programming fundamentals, commands descriptions, ST graphics & sound, using GEM in BASIC, file management, disk operation. Tutorial problems give hands on experience. 300pp $16.95

BASIC to C
Move up from BASIC to C. If you're already a BASIC programmer, you can learn C all that much faster. Parallel examples demostrate the programming techniques and constructs in both languages. Variables, pointers, arrays, data structure. 250pp $19.95

3D GRAPHICS
FANTASTIC! Rotate, zoom, and shade 3D objects. All programs written in machine language for high speed. Learn the mathematics behind 3D graphics. Hidden line removal, shading. With 3D pattern maker and animator. $24.95

The ATARI logo and ATARI ST are trademarks of Atari Corp.

Abacus Software

P.O. Box 7219 Dept. A9 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

Optional diskettes are available for all book titles at **$14.95**

Call **now** for the name of your nearest dealer. Or order directly from ABACUS with your MasterCard, VISA, or Amex card. Add $4.00 per order for postage and handling. Foreign add $10.00 per book. **Other software and books coming soon.** Call or write for your **free** catalog. Dealer inquiries welcome–over 1400 dealers nationwide.

ATARI® ST™

Introduction to MIDI Programming

MIDI –The Musical Instrument Digital Interface. The Atari ST has a built-in MIDI interface, which means that it's ready to hook up to any digital electronic musical instrument equipped with MIDI ports. The ***ST Introduction to MIDI Programming*** provides the groundwork for discovering the infinite musical possibilities of the Atari ST's MIDI interface and your synthesizer. Topics include:

- Introduction to MIDI programming
- MIDI STANDARD and MIDI LANGUAGE
- Programming your synthesizer
- How to buy a synthesizer
- How to buy MIDI software
- Using the extended BIOS
- The source code from Xlent Software's ST MUSIC BOX AUTO-PLAYER program
- C source codes for a number of programs and functions

About the authors:
Len Dorfman and Dennis Young have written a number of best selling Atari 8-bit & 16-bit programs for XLent Software. They have combined their programming talents and musical knowledge (Len Dorfman plays jazz guitar) to introduce you to the MIDI interface of the Atari ST.

ISBN 0-916439-77-1

The ATARI logo and ATARI ST are trademarks of Atari Corp.

you can count on
Abacus

ATARI ST INTRO TO MIDI P $7.00
2320042921650 DORFMAN USED 04
POWELL'S 04 COMP-ATARI MU